ASPEN

AUTOMATED SCHEDULING AND PLANNING ENVIRONMENT

User's Guide

Version 2.0

Technical Document D-15482
02/28/00 3:00 PM

Document Written by:

Robert Sherwood
Barbara Engelhardt
Gregg Rabideau
Steve Chien
Russell Knight

Artificial Intelligence Group
Information and Computing Technologies Research Section
Information Systems Development and Operations Division
Jet Propulsion Laboratory
California Institute of Technology

Table of Contents

1. Introduction

ASPEN is an object-oriented system that provides a reusable set of software components that implements the elements commonly found in complex planning/scheduling systems. These components include:

- An expressive constraint modeling language to allow the user to define naturally the application domain
- A constraint management system for representing and maintaining spacecraft operability and resource constraints, as well as activity requirements
- A set of search strategies
- A temporal reasoning system for expressing and maintaining temporal constraints
- A graphical interface for visualizing plans/schedules (for use in mixed-initiative systems in which the problem solving process is interactive

The primary application for ASPEN has been the spacecraft operations domain. Planning and scheduling spacecraft operations involves generating a sequence of low-level spacecraft commands from a set of high-level science and engineering goals. ASPEN encodes spacecraft operability constraints, flight rules, spacecraft hardware models, science experiment goals, and operations procedures to allow for automated generation of low-level spacecraft sequences.

1.1. Activity Database

The central data structure in ASPEN is an activity. An activity represents an action or step in a plan/schedule. An activity has a start time, end time, and a duration. Activities can use one or more resources. All activities in a plan/schedule are elements of the Activity Database (ADB), which maintains the state of all of the activities in the current plan/schedule, and serves as the integrating component that provides an interface to all of the other classes.

One function of the ADB is to represent and maintain hierarchical relationships between activities. Activities can contain other activities as decompositions; this facility can be used to reason about the plan/schedule at various levels of abstractions (e.g., a scheduler can first reason about a set of activities without considering that each of those activities are themselves composed of a set of decompositions – this can make various reasoning tasks much more computationally tractable.

Temporal and resource-constraints between activities are also represented in the ADB. Although most of the actual computational mechanisms that maintain these constraints are implemented in other the modules described below, the protocol that a search algorithm uses to access constraints in the context of a plan/schedule is implemented in the ADB. For example, although the resource timelines are responsible for detecting overuse of resources by activities, the ADB maintains data structures that indicate the assignment of activities to specific timelines, so that one can efficiently ask queries such as, "which resources does this activity use?"

As another example: although the temporal constraint network is responsible for maintaining temporal constraints between individual activities, the ADB is responsible for global constraints. (e.g., the ADB contains global constraints such as: "all activities occur after the start of the scheduling horizon." When an activity is created, the temporal constraint that the activity occurs after the horizon is created automatically by the ADB).

1.2. Temporal Constraint Network

A Temporal Constraint Network (TCN) is a graph data structure that represents temporal constraints between activities. A temporal constraint describes the temporal relationship between an activity and other activities and/or the scheduling horizon, and imposes an ordering on the set of activities. The TCN implements a Simple Temporal Problem, as defined in [3], and represents a set of constraints, all of which must be satisfied at any given time, i.e., it represents the conjunct of all active constraints between activities in the ADB. Activities are represented in the TCN as pairs of time points, where each time point corresponds to the beginning or end of an activity, and the edges in the TCN graph represent the constraints on the temporal distance between the time points. The TCN can be queried as to whether the temporal constraints currently imposed between the activities are consistent or not (i.e., whether the current ordering is free of cycles).

1.3. Resource Timelines

Resource timelines are used to reason about the usage of physical resources by activities. Capacity conflicts are detected if the aggregate usage of a resource exceeds its capacity at any given time. Several subclasses of resource timelines are implemented, including depletable resource timelines used to model consumable resources (e.g., fuel), and non-depletable resources that are used to model resources which are not actually consumed by usage, but are instead “reserved” for a period of time (e.g., a piece of equipment). Our current model of resource usage is discrete. That is, if we specify that an activity such as `move_forward` uses two units of fuel, then both of these units are modeled as being immediately consumed at the beginning of the activity. This is a discrete approximation, since the usage of the fuel may be better modeled as a linear function such as usage(t) =2t/(activitylength), where t is the time elapsed since the beginning of the activity, and activitylength is the duration of the activity.

1.4. State Timelines

State timelines represent arbitrary attributes, or states, that can change over time. Each state can have several possible values; at any given time, a state has exactly one of these values. Activities can either change or use states. For example, a door-open activity would set the state of door to be open, while an enter-building activity would require that the state of door be open. As activities are placed/moved in time, the state timeline updates the values of the state, and detects possible inconsistencies or conflicts that can be introduced consequently. For example, an activity that requires that the door be open is placed at time t, then the state timeline checks to verify that the door is in fact open at time t. Otherwise, a state constraint violation is indicated. Users can define legal sequences of state transitions. The state timeline class will detect illegal transition sequences if they are introduced into the timeline.

For example, consider modeling a traffic light with a state timeline, traffic-light. The possible values are green, yellow, and red. The legal state value transitions are: green to yellow, yellow to red, red to green. All other transitions are illegal.

1.5. Parameter Constraint Network

Each activity has a number of parameters that are either user-defined or computed by the system, such as start time, end time, duration, any resources it uses, any states it changes/uses, etc. In ASPEN, it is possible to create dependencies between pairs of parameters within the same activity, or between pairs of parameters defined in different parameters. A dependency between two parameters p1 and p2 is defined as a function from one parameter to another, p1 = f(p2), where f(x) is an arbitrary function whose input is the same type as p2, and whose output has the type of p1. A dependency is a constraint on the values of a parameter. These dependencies are represented and maintained in a Parameter Constraint Network (PCN). The PCN maintains all dependencies between parameters, so that at any given time, all dependency relations are satisfied. Note that if there exists a dependency such that p1 = f(p2), its inverse dependency, p2 = f^{-1}(p1) does not necessary exist, unless the user specifies the inverse relationship and specifies the inverse dependency as well.

Note that the TCN can be seen as a special case of a PCN in which the functional relationships between the parameters (activity start/end times and durations) is a distance relationship, and for which very efficient constraint propagation algorithms have been implemented. In general, as commonly used special cases of functional dependencies between parameters (such as temporal distance relationships), it can be useful to develop special dependency networks that implement efficient constraint propagation algorithms that take advantage of the special structure of these dependencies, instead of relying on the general mechanism offered by the PCN. Such special-purpose dependency networks can be implemented as subclasses of the abstract PCN, or (if the protocol that must be supported is sufficiently unique) abstracted out as a separate basic component of ASPEN, as was done with the TCN.

1.6. Planning/Scheduling Algorithms

The search algorithm in a planning/scheduling system searches for a valid, possibly near-optimal plan/schedule. The ASPEN framework has the flexibility to support a wide range of scheduling algorithms, including the two major classes of AI scheduling algorithms: constructive and repair-based algorithms.

Constructive algorithms incrementally construct a valid schedule, taking care that at every step, the partial schedule constructed so far is valid. When a complete schedule is constructed, it is therefore guaranteed to be valid. Repair-based algorithms generate a possibly invalid complete schedule using either random or greedy techniques. Then, at every iteration, the scheduled is analyzed, and repair heuristics that attempt to eliminate conflicts in the schedule are iteratively applied until a valid schedule is found.

The following repair-based search algorithms have been implemented in ASPEN:

- Depth-first - a constructive, exhaustive, backtracking algorithm
- Repair - a extension of the DCAPS repair-based algorithm
- Hill-climbing - only make repair operations that give you better schedules (where 'better' is user-defined)
- Simulated annealing - similar to hill-climbing, but makes random repair operations with some decreasing probability
- Balanced beam - beam search that tries to balance the search tree
- 3-deep beam - beam search that goes at most 3 levels deep
- Iterative deepening A* - A* search that iteratively tries deeper searches

In addition, two initial schedule generation functions have been implemented. One, called Simple Placement, just grounds activity start times at the earliest time and places all activities on their timelines (i.e. simulates the affects of the activities on the states and resources). The other, called Forward Dispatch, first expands the activities, then places them on the timelines at legal locations in a forward-sweeping manner.

Both algorithms may not resolve all conflicts, since possible violations of temporal and resource/state constraints may not have been considered. Any of the repair algorithms can be used to solve many of the remaining conflicts.

2. Modeling

The ASPEN modeling process in outlined in Figure 1.

1. Create a sub-directory of the models/ directory to store model files
2. Create model.mdl file defining high level model characteristics
3. Create activities.mdl file defining the domain activities
4. Create resources.mdl file defining the resources used in the activities
5. Create states.mdl file defining the state variables used in the activities
6. Create parameters.mdl file defining any parameters used in the activities
7. Create timelines.ini file with instances of resources and states
8. Create init-state.ini file with instances of activities to model during the planning horizon

Figure 1 Modeling Summary

The ASPEN modeling language (AML) allows the user to define activities, resources, and states as well as any constraints between them. A domain model is input at start-up time, so modifications can be made to the model without requiring ASPEN to be recompiled. The modeling language has a simple syntax, which can easily be used by non-AI personnel (e.g. spacecraft mission operations personnel) to create a model. Much of the syntax will look similar to the C programming language. Each model (e.g. spacecraft model) is comprised of several plain-text files, which define and instantiate activities, resources, and states. The overall modeling process is summarized in Figure 1 Modeling Summary. The modeling process in ASPEN involves defining the planning domain using the AML. The domain itself is specified in the `model.mdl` file. An example of this file in contained in Figure 2. The model file contains the model name (`mars_pathfinder`), the start time for the current planning period, and the duration of the planning period. Each of these parameters is required for ASPEN to run. The start time is specified in the given time format. All relative times are interpreted as a number of centiseconds, seconds, minutes, hours, days or weeks since the horizon start. The end time for the planning period can also be specified using the "`horizon_end`" expression (this can be used in place of a duration, otherwise it must be consistent with the duration).

```
model mars_pathfinder {
      time_scale = second; // can be centisecond, second,
                           //minute, hour, day, week
      horizon_start = 1998-185/12:00:00;
      horizon_duration = 7d;  // 7 days
      time_zone = local; // can be local or utc
      time_format = slash; // can be slash, tee, or calendar
};
```

Figure 2 Model.mdl File Example

The “`time_scale`“ expression is used to set the units for all relative time specifications. If no time scale is specified, the default is second. A smaller time unit than the time scale for the model cannot be specified in the model. The time scale can be set to centisecond, second, minute, hour, day, or week. The time scale should be the first parameter specified in the model structure, so that the rest of the parameters are scaled appropriately. The horizon duration, like all times in ASPEN, can be specified in terms of centiseconds, seconds, minutes, hours, days, or weeks. The following list contains the syntax for these time specifications:

- <int> - number of time units (default to seconds)
- <int>s – number of seconds
- <int>m – number of minutes
- <int>h – number of hours
- <int>d – number of days
- <int>w – number of weeks

The <int> represents any integer value. For example, “`horizon_duration = 2h`“ represents a duration of two hours.

The “`time_zone`“ expression is used to set the time zone for all times specified in ASPEN. The local time zone is used by default. If the value is “`local`,” all time strings will be interpreted and represented as local times determined by the running computer system. If the value is “`utc`,” all times will be encoded with respect to Universal Time Coordinated (UTC). This is the same as Greenwich Mean Time (GMT).

The “`time_format`“ expression is used to set the format of the time strings. This will be the assumed format for reading and writing all time strings. The default time format is “`slash`.” If the value is “`slash`“ the time strings must be of the form: `YEAR-DOY/HH:MM:SS` where `DOY` is the numerical day-of-year (number of days since January 1) and `HH:MM:SS` is hours, minutes, seconds in 24 hour time. The “`slash`” refers to the separator between the day-of-year and the hours. If the value for “`time_format`“ is “`tee`,” the time strings must be the same as “`slash`“ but with a ”T” separating the day-of-year and hours (`YEAR-DOYTHH:MM:SS`). If the value is “`calendar`,” the time strings will be of the form: `YEAR MON DAY HH:MM:SS` where `MON` is the three letter abbreviation for the month and `DAY` is the day of the month. If the time scale for the model is centiseconds, all of these time formats can include a decimal point followed by two digits representing the centiseconds (`YEAR-DOY/HH:MM:SS.CS,` for example).

The next step in modeling the domain involves defining the activities to be planned, followed by the resources, and state variables that are used by those activities. The modeling language has a simple syntax, which can easily be used by non-AI personnel to create a model. Each model is comprised of several plain-text files such as an `activities.mdl` file, etc. Activities, resources, and states are defined in `.mdl` files and then instantiated in `.ini` files. AML uses a C like syntax with braces around definitions and semi-colons at the end of lines. Comments are defined using the C style (`/* comment */`) or the C++ style (`// comment`). The domain model is an input to ASPEN, generally at start-up time, so modifications can be made to the model without requiring ASPEN to be recompiled.

2.1. Activities

As previously mentioned, activities are the central data structure of ASPEN. An activity is a data structure that represents an action in the planning domain. Activities are generally defined in the `activities.mdl` file. Figure 3 contains examples of an instrument data take activity and a solar array drive state change activity. These examples will be used to explain the components of an activity.

```
Activity ALI_data_take {
   string comment = “data collection with the Advance Land Imager (ALI)”;
   int power = 100; // 100 watts
   duration = [1,60];
   constraints =
      starts_after end of SAD_changer with (“fixed”->sad1) by [100,300] and
      ends_before start of SAD_changer with (“track”->sad1) by [16,16] and
      contains all of SAD_user with (“fixed”->ap1) by [4,4,0,10] and
      contained_by all of SAD_user with (“fixed”->sad1) by [300,300,1,16];
   decompositions = ALI_user_data, ALI_dark_count;
  reservations = processor, array_power = 80, bus = 2;
12  };
13
14  Activity SAD_changer {
15     sad_mode sad1;
16     reservations = SAD_sv change_to sad1;
17                    aperture must_be open;
18  };
```

Figure 3 Activity Example

Definition/Type

An activity is defined in line 1 and line 14 of Figure 3. The definition includes the keyword "`activity`," the user-defined activity name, and the activity body (enclosed in braces and followed by an optional semi-colon). These are the only required components of an activity definition. Once the activity is defined, it can be instantiated in the initial state file. Generally, this instance will consist of the activity name followed by the instance name and the instance body (enclosed in braces). The components of the activity body are described below. Many of them can be specified with undetermined values that can be assigned specific values in the activity instance.

Parameters

All parameters must be declared first in the activity body. There are four types of parameters: integer (`int`), string (`string`), boolean (`bool`) and floating point (`real`). Parameters are declared with one of the four internal types, or with a user-defined type specified, for example, in the `parameters.mdl` file. Defining global user types is described in a later section. Inside an activity, parameters are declared by first giving the type identifier, then the parameter name, and an optional assignment. The parameter declarations must reside before assignments (including duration) in an activity.

Parameters are generally used to store values in activities or reservations. Lines 2 and 3 of Figure 3 show parameters declared using the internal types. In this case, they are both assigned constant values. Line 15 shows a parameter declared from a user-defined type. Parameters can also be passed into activities from higher or lower level activities (parent/child activities) or from activities connected through constraints or decompositions. Lines 6-9 contain examples of parameters that are passed from one activity to another. The parameter "`sad_mode`" on line 15 is an enumerated type variable whose value can be one of "`fixed`" or "`tracking`". Line 6 shows the value "`fixed`" being passed to the value of "`sad1`" in activity `SAD_changer` when it is a subactivity of `ALI_data_take`. These are called parameter dependencies and are described in a later section.

Start Time, End Time and Duration

Start time, end time and duration are internal activity parameters and do not need to be declared since they exist for all activities. However, like any other parameter, each can be assigned a range `[x,y]`, a list `(a,b,c,d)`, or a constant. All internal parameter assignments must occur after the parameter declarations. Line 4 assigns the duration to a range of 1-60 seconds, given that the time scale, specified in the `model.mdl file, is seconds`. All ranges within ASPEN can be specified from negative infinity to positive infinity. Start time and end time must be in the range from zero to infinity. Duration must be from one to infinity. These are also the default values for a parameter if no assignment is given. The keyword "`INFINITY`" is a reserved word within ASPEN. The duration is a relative time and should be specified in terms of centiseconds, seconds, minutes, hours, days, or weeks. For example, "`duration = 2h`" represents a duration of two hours. If a range is given for the duration, ASPEN will have more flexibility in considering different schedules. This can result in better-optimized schedules. The start- and end-time, and duration ranges of an activity can be assigned with the keywords “start_time“, “end_time,” and “duration.” If the end time is specified, it must either replace or be consistent with duration.

Decompostion Index and Priority

Decomposition index and priority are internal activity parameters just like start time, end time, and duration, and do not need to be declared. They can also be assigned an integer value (decomposition index must be non-negative), a range, or a list, and assignment must occur after the parameter declarations. The parameter “decomposition_index” refers to the index of the decomposition when a decomposition is specified (described later), and the range is {0, n-1} where n is the number of decompositions in an activity. The parameter “priority” refers to the priority of an activity. The range of priority is currently (-Infinity, Infinity) with the lower integers having lower priority. Priority is used in preemption, which is described later, but can also be used in repair and optimization heuristics or user scheduling functions.

Temporal Constraints

The "constraints" in an activity type definition are temporal constraints on all activities of that type. If it exists, the constraint assignment must occur after all parameter declarations. If no constraints are specified, then the activity will not have any temporal restrictions (other than the default constraint requiring it to occur within the planning horizon). A constraint requires at least one activity of the given type to exist with the given temporal relation. For example:

```
Activity A {
  constraints = starts_after end of B by [10, 20];
};
```

This constraint says that *all* activities of type A require at least *one* activity of type B to end before the start of the type A activity. In addition, the “by” expression specifies the minimum and maximum separation distance between the two time points. In this example, the distance between B's end and A's start must be at least 10 units and at most 20 units (where the unit is determined by the time scale). Therefore, the following is valid:

```
 B1             A1     A2
|---|          |---|  |---|

..........................
1   5          15     20
```

Figure 4 Constraint Example

In Figure 4, A1 starts 10 units after B1 and A2 starts 15 units after B1, so this is a possible final schedule. Any activity of type B can satisfy one or more of the temporal constraints of activities of type A. If B1 is deleted, another one or more activities of type B will have to be found to satisfy the constraints of A1 and A2. If A1 and A2 are deleted, so are their constraints, so B1 may no longer be needed. However, it's not necessarily deleted because another activity may require it.

There are eight types of constraints: starts_before, starts_after, starts_at, ends_before, ends_after, ends_at, contains, and contained_by. The starts_before, starts_after, ends_before, ends_after, constraint types include a time range and a temporal relationship to the start of or end of the activity in question. For example, on line 6 in Figure 4, the ALI_data_take activity must start after the end of the SAD_changer activity by 100-300 seconds. This constraint tells the scheduler that the SAD_changer activity must be completed at least 100 seconds and at most 300 seconds before the ALI_data_take activity is initiated. Using the same method, the start- or end-time of any activity can be specified relative to the start or end time of the owner activity. If a distance is specified as [0,0], the start or end times will coincide exactly. If no distance is given, then the default distance range is from zero to infinity. The `starts_at`, `ends_at` types have the same functionality as the earlier types except that the relative start- or end-times are 0.

The "`contains`" constraint is used for activities that fall within the temporal scope of the owner activity. This constraint definition combines a "`starts_before start`" and an "`ends_after end`" pair of constraints. For example, line 8 of Figure 3 defines a constraint for the `SAD_user` activity that is contained within owner activity `ALI_data_take`. The first two and last two numbers in the constraint represent ranges of time, which separate the start times between the two activities and the end times between the two activities. `SAD_user` must start exactly four seconds (4,4) after the start of `ALI_data_take` but the end time can coincide with the end time of `ALI_data_take` or up to 10 seconds (0,10) earlier. This relationship is graphically represented in Figure 5. Because the `ALI_data_take` activity has a variable duration, ASPEN automatically chooses a duration that satisfies the above temporal constraints.

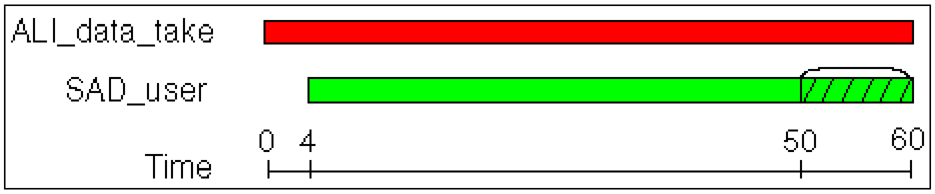


Figure 5 Constraint Relationship: contains both of

The "`contained_by`" constraint is used for activities that are the same size or larger than the owner activity. For example, line 9 defines a constraint for activity `SAD_user` that starts exactly 300 seconds before the start of and ends 1-16 seconds after the end of activity `ALI_data_take`.

Multiple constraints to various activities are combined with ANDs and ORs to make the constraint expression for the given activity. Parentheses can also be used to group the constraints. In this way, arbitrary AND-OR constraint trees can be constructed for an activity.

In the "by" expression, the ranges can be replaced by integer range parameter names. For "`contains`" and "`contained_by`," two parameters are required. All others require only one. For example:

```
Activity A {
  int x = [10, 20];
  int y = [20, 30];
  int z = [30, 40];
  constraints = starts_after end of B by x,
    contains all of C by y,z;
};
```

Temporal constraints allow the use of the "`with`" expression (shown on lines 6-9 of Figure 4) to pass parameters. The "`with`" expression is optional and, if exists, follows the name of the external activity that has the parameters being passed. This consists of the keyword "`with`" followed by a list of parameter dependencies. Instead of using an implicit ordering of the parameters, ASPEN allows the user to specify any subset of parameters in any order. This has the disadvantage that you need to explicitly name the parameters you are referring to, but there is no confusion on which parameters are being passed. The left side of the arrow is a parameter name from the activity in the current scope and the right side is a name (or list of names) of a parameter from an external activity. The arrow can be in either direction and can have a function associated with it (implicitly, no specified function is the identity function). The parameter receiving a value or set of values must be capable of taking on that value or set of values. On line 6 of Figure 4, the value "`fixed`" is propagated to the parameter "`sad1`" of the connected `SAD_changer` activity. Internal parameters such as duration can also be propagated. Parameter dependencies are discussed in more detail in a later section.

Avoid having "loops" in your model. Don't have one activity require another activity that eventually requires the first activity. For example:

```
Activity A {
  constraints = starts_after end of B;
};

Activity B {
  constraints = ends_before start of A;
};
```

Although this is technically possible, it will make things difficult for the automated schedulers by allowing the possibility of looping forever.

Decompositions

Decompositions (i.e. subactivities with constraints) are sets of activities that can be scheduled any time strictly within the duration of the parent activity subject to temporal constraints. The decomposition assignment must be defined after declarations in activities. A decomposition declares one possible expansion for an abstract activity. With multiple disjunct decompositions, only one needs to be satisfied at any given time. Line 10 defines a single decomposition into subactivities `ALI_user_data` and `ALI_dark_count`. These activities must fall within the temporal duration of the parent activity `ALI_data_take`. The main difference between "`constraints`" and "`decompositions`" is that constraints can be satisfied by any activity in the schedule. New activities are always created for each subactivity in a decomposition. There is a one-to-one relationship from parents to subactivities. For example:

```
Activity A {
  decompositions = B;
};
```

This decomposition says that all activities of type `A` must contain a unique activity of type `B`.

In decompositions, there is an implicit temporal constraint between the parent and it's children that requires the children to occur within the duration of the parent (i.e. contains). In addition, activities *cannot* share subactivities. Therefore, in a feasible schedule, subactivities cannot be deleted without deleting the parent, and a parent activity cannot be deleted with deleting all the children. Creating and deleting subactivities is done through “detailing” and “abstracting” activities. In addition, there is a tighter coupling with subactivities than with constraints. Subactivities are like macro expansions, where as constraints have more flexibility. More flexibility may allow more optimal schedules, but may cause the automated schedulers to take more time to find a schedule. For example, in Figure 4, the scheduler has to decide which activity of type `B` to use to satisfy the constraints for `A1` and `A2`, whereas in decompositions, there's no choice - new subactivities are always created.

The order of decomposed activities can be defined with a “`where`” expression. The keyword “`where`” is followed by the keyword “`ordered`” or by an AND-OR constraint tree similar to what was described in the previous section. The only difference being that both activities participating in the constraint must be specified (the source is no longer implicitly the activity in context). The keyword "`ordered`" is used to specify the decomposed activities to be scheduled in sequential order (i.e. each ends before the start of the next).

The "`with`" expression described in the previous section can also be used in decompositions.
For example:

```
Activity A {
  int a = 10;
```

```
          decompositions = B with (a->b);
        };
```

where b is a parameter of activity type B, and the parameter type of b must include the value 10.

An activity could have several possible decompositions. For example, if A can consists of B and C or D and E, in that order, then A can be specified as follows:

```
        activity A {
          decompositions = (B,C where B starts_at start of C) or
                           (D,E where D starts_before start of E);
        };
```

The resulting schedule will either contain an A, B, and a C, or it will contain an A, D, and an E.

Decompositions can consist of more than one subactivity of the same type. Constraints among subactivities in the context of the parent activity are defaulted to be for-all/for-all. For example:

```
        activity A {
          decompositions = (B,B,C,C where B starts_before start of C) or
                           (D,D,E where D starts_before start of E) or
                           (F,G,G where F starts_before start of G);
        };
```

The first decomposition implies both B's have to start before both C's. The second implies both D's have to start before the E. The third implies the F activity has to start before both G's. But, if A consists of two B's and one C, and C has to be in between the two B's, then it may also be defined as follows:

```
        activity A {
          decompositions = B b1,B b2,C where b1 starts_before start of C,
              C starts_before start of b2;
        };
```

By labeling the subactivities, you can specify which subactivity to choose for the constraints with the label, and the constraint is no longer for-all/for-all. Another example:

```
        activity A {
          decompositions = (B b1,B b2,C,C
              where b1 starts_before start of C,
                    C starts_before start of b2) or
            (B b1,B b2,C c1,C c2
              where b1 starts_before start of C,
                    C starts_before start of b2) or
            (B b1,B b2,C c1,C c2
              where b1 starts_before start of c1,
                    c2 starts_before b2);
        };
```

The first implies both C's have to be in between b1 and b2 where b1 is before b2. The second implies the same as the first. In the third statement, b1 starts before c1, and c2 starts before b2. ASPEN does not allowed OR's in internal subactivity constraints.

Constraints can also be made between the parent activity and one or more of the subactivities. In the “where” expression, simply specify the name of the parent as one of the activities of the constraint. For example:

```
        activity Z {
          decompositions = B,C where Z starts_before start of B by [5,5],
              C ends_before end of Z by [10,20];
```

```
      };
```

This decomposition requires that the B subactivity starts exactly 5 units after the start of parent activity Z and that the C subactivity ends 10 to 20 units before the end of parent activity Z. Keep in mind that there is always an implicit "contains" constraint between a parent and its children that must be consistent with any addition constraints in the "where" expression.

ASPEN maintains an internal parameter that represents the current decomposition of an activity. This parameter (called "decomposition_index") only exists for activities that have decompositions. It is a range of integers with allowable values from 0 to one less than the number of possible decompositions and represents the index of the decomposition in the order they are defined. In the example above, there are three decompositions for activity A, so the allowable range of its "decomposition_index" parameter is [0,2]. Assigning a value to "decomposition_index" will detail the activity to the corresponding decomposition. For example, given the following partial model definition, consider an instance of parent_foo pf1.

```
      activity parent_foo {
        int my_index = [0, 1];
        decompositions = (child_a) or (child_b);
        dependencies = decomposition_index <- my_index
      };
```

If my_index in pf1 were to be ground to 1, decomposition_index would be assigned a value of 1, and pf1 would be detailed using the second decomposition giving it a child of type child_b. In this way, the modeler can implement decompositions that are determined by parameters and dependency functions.

Reservations

Reservations are used to reserve a portion of a resource or state for the duration of the activity. There are two types of reservations: state and resource. Resource reservations can be further broken down into two types: atomic and aggregate. Line 11 of contains examples of an atomic reservation (processor) and an aggregate reservation (array_power). The processor reservation reserves the processor for the duration of the activity. No other activities can use the processor during this time. The array_power reservation uses 80 units of the array_power resource. If the total capacity of the aggregate resource at the time of the reservation is greater than 80, then additional activities can reserve the remaining power. Negative values can be reserved on depletable aggregate resources in order to replenish the resource, such as replenishing power for a battery or freeing up memory by downlinking stored data. Additional properties of resources will be discussed shortly.

Resource reservations are defined in three different ways. For atomic resources, only the resource name is given. For aggregate reservations, the resource and amount used must be specified. Table 1 lists examples of resource reservations.

Syntax	**Notes**
`reservations = IR_camera;`	Atomic resource
`reservations = solar_array_power use 50;`	Aggregate resource
`reservations = solar_array_power use -50;`	Aggregate, depletable resource
`reservations = solar_array_power = 50;`	Aggregate resource
`reservations = solar_array_power = SApower;`	SApower is a parameter

Table 1 Resource Reservation Examples

State reservations either require a particular value of a state variable or change the value of a state variable. The monitoring type reservation ensures that the state variable will not change value during the activity. The keyword for the monitoring type reservation is “must_be,” and this requires that the state variable has the monitored value for the entire duration of the activity. The keyword for a changing reservation is “change_to.” The point at which the change occurs can also be specified with the “at_start” and “at_end” keywords. By default, the state change occurs at the start of the activity. If “at_end” is specified, the change will occur at the end of the activity, or both can be used to change the state variable to one state at the beginning of an activity and change it to another state at the end of the same activity. Table 2 lists examples of state reservations. Figure 3 has additional examples of reservations within an activity. Line 16 of Figure 3 has a reservation that changes the state of the SAD_sv state variable to the value of parameter sad1. Line 17 of Figure 3 reserves the "open" state of the aperture state variable for the duration of the activity. If the aperture state variable is in a state other than "open" during this activity, the scheduler would need to find a state changer activity to change the state to "open" prior to the activity’s start time.

Syntax	**Notes**
`Reservations = instrument_sv must_be “idle”;`	State monitoring type reservation
`Reservations = instrument_sv change_to “on”;`	State changing type reservation
`Reservations = instrument_sv change_to mode;`	mode is a parameter

Table 2 State Reservation Examples

Permissions

The permissions of an activity are used by the scheduling algorithms to permit/prevent certain operations on an activity. The permissions statement has the following syntax:

```
permissions = (“move", “add", “delete", “detail”, “abstract”,
      “place”, “remove”, “connect”, “disconnect”, “parameters”,
      “duration”);
```

If no permissions statement is included, the default is all. Table 3 lists the description of the operations defined in permissions.

Operation	**Description**
move	Allows the scheduler to change the start-time, duration, and end-time of an activity
add	Allows the scheduler to add new instances of the activity
delete	Allows the scheduler to delete the activity
detail	Allows the scheduler to detail the activity into subactivities
abstract	Allows the scheduler to abstract the activity, deleting subactivities
place	Allows the scheduler to place the activity on the timelines
remove	Allows the scheduler to remove the activity from the timelines
connect	Allows the scheduler to connect the temporal constraints of the activity
disconnect	Allows the scheduler to disconnect the temporal constraints of the activity
parameters	Allows the scheduler to set parameter values of the activity
duration	Allows the scheduler to change the duration of the activity

Table 3 Permission Operations

An additional statement, no_permissions, is used to prevent the repair algorithm from performing certain operations on activities. Any operations not listed in the no_permissions statement will be allowed. An example where this would be useful is an activity that sets a state variable to either daytime or nighttime. Because this is an environmental state variable, it is not something that can be planned or modified. In this case, the user would not want ASPEN to move, add, or delete this activity. The following no_permissions syntax accomplishes this goal:

```
      no_permissions = (“move”, “add”, “delete”);
```

The user should not use a `permissions` and `no_permissions` statement in the same activity. Permissions can be set in either the activity definition (model file) or the activity instantiation (.ini file).

Activity Instances

After activities are defined in the model file, they must be instantiated in initial state/request files (`.ini`). Generally an activity instantiation consists of the activity name, an instance name, a start time, and any parameter assignments that the user wants to set an initial value for in the activity. An activity can be instantiated as many times as required. Figure 6 contains an example of an instantiated activity.

```
ALI_data_take adt1 {
  start_time = 1998-148/12:30:00;
};
```

Figure 6 Instantiated Activity Example

The assigned values for start time, end time, or any parameter are just initial values, and they may be changed during the planning process. In order to constrain these values for the particular activity instance, an ampersand can be used with the equal sign as in Figure 7 for the duration parameter. Another way of constraining the values is to set permissions of the activity to disallow changes to parameter values.

Mandatory and Optional Goals

When activities are instantiated in initial state/request files (`.ini`), the user can specify whether the activity instances are mandatory or optional goals. If the instance is a mandatory goal, then it is immediately placed in the schedule at the appropriate start time. If it is removed from the schedule, a conflict is generated. This means that it is possible to delete a mandatory goal from a schedule, but the resulting schedule will be infeasable. Inorder to create a feasible schedule, the schedule must contain all of the mandatory goals in the .ini files.

```
ALI_data_take adt3 mandatory_goal {
  start_time = 1998-148/12:32:30;
  duration =& 26s;
};

ALI_data_take adt4 optional_goal {
  start_time = 1998-148/12:35:00;
  duration =& 28s;
};
```

Figure 7 Mandatory and Optional Goal Example

Optional goals are not placed in the schedule when the .ini file is loaded. A feasible schedule can be created without including one optional goal. Instead, optional goals are not placed until the optimization algorithm is run and there is a preference for more optional goals in the `preferences.tab` file. The optimization algorithm will place an optional goal to attempt to optimize the preference for more optional goals. If this fails, the schedule may be feasible without the optional goal, but it cannot be optimal.

Other than the mandatory or optional goal specification, the activity instances that are goals are the same as the non-goal instances.

Activity Request Language

Activities that occurs at regular intervals during the planning horizon can be instantiated using the AML request language. The format for these requests in the initialization file is:

```
<activity type> occurs [<int> times] every <rel_time>
[starting <abs_time>] [ending <abs_time>]
```

Where:

`[<int> times]` is the number of times to repeat the activity
`<rel_time>` is the time period over which you are repeating the activity
`starting` and `ending` are the start and end times over which the repetition will occur

Each keyword surrounded by brackets is optional. Some example of the request language are contained in Table 4.

Request Syntax	**Description**
`act1 occurs every 5s starting 1998-240/08:00:00;`	Repeat activity act1 every 5 seconds starting at 1998-240/08:00:00 until the end of the planning horizon
`act2 occurs every 10m;`	Repeat activity act2 every 10 minutes during the entire planning horizon
`act3 occurs 4 times every 1d;`	Repeat activity act3 four times a day during the entire planning horizon

Table 4 Request Language Examples

2.2. Resources

A resource is a description of a profile of a physical resource over time. There are four types of resources: atomic, concurrency, depletable, and non-depletable. Atomic resources are physical devices that can only be used (reserved) by one activity at a time. Examples of atomic resources include: science instrument, star tracker, reaction wheel, or CPU. Concurrency resources are similar to atomic except they must be made available to the activity before they are reserved. An example would be a tele-communications downlink pass. The telecommunications station would have to be made available before the spacecraft could initiate a downlink. Non-depletable resources can used by more than one activity at a time and do not need to be replenished. Each activity can use a different quantity of the resource. Examples include solar array power and the 1773 bus. Depletable resources are similar to non-depletable except that their capacity is diminished after use. In some cases their capacity can be replenished by other activities during a mission (battery energy, memory capacity) and in other cases it cannot (fuel). A summary of the three types of resources is presented in Table 5. Resources are instantiated in a `.ini` (usually `timelines.ini`) file similar to activities. All resources must have exactly one instance defined.

Resource Type	Properties	Examples
Atomic	Always available when not in use, only 1 user at a time	science instrument, star tracker, reaction wheel, cpu
Non-depletable	Always available when not in use, many users can use different quantities	solar array power and 1773 bus
Depletable	Capacity is diminished after use, may or may not be replenished by another activity, many users	battery energy, memory capacity, fuel

Table 5 Resource Types

Definition/Type

Resources are defined in the `resources.mdl` file. The three types of resources are defined in lines 1, 5, and 10 of Figure 8. The definition includes the name followed by a pair of braces and a semi-colon similar to the Activity definition. The type is one of: atomic, depletable, and non-depletable. The name and type are the only required components of a resource definition. Once the resource is defined, it can be instantiated exactly once in the initial state file. Generally, this instance will consist of the resource name followed by the instantiated name and a pair of braces.

```
1   Resource ALI {
2     Type = atomic;
3   };
4
5   Resource Solar_array {
6     Type = non_depletable;
7     Capacity = 600; // watts
8   };
9
10  Resource warp_storage {
11    Type = depletable;
12    Capacity = 40000;  // Mbits
13  };
```

Figure 8 Resource Examples

Capacities

The minimum value, default value, and the capacity of a resource can be specified using the `min_value, default_value,` and `capacity` keywords. All values must be integers. ASPEN does not accept units for resource values (in the example above, they are just in comments) and therefore does not do calculations with different metric units. The default capacity (if one is not given) is infinity, the default min value is zero, and the default value is zero. An atomic resource has a unit capacity and therefore does not accept the capacity keywords. Depletable and non-depletable resources definitions can have a capacity such as in lines 7 and 12 of Figure 8.

2.3. States

A device, subsystem, or system may be represented by a state variable that gives information about its state over time. The state variable contains the current state that is defined as an enumerated type or vector. Some examples of possible states are on, off, open, closed, record, playback, standby, or idle. States can be reserved or changed by activities. A state variable must equal some state at every time. At the beginning of a planning horizon, this state is just the default state. Figure 9 contains two examples of state variable definitions. State variables are defined in the `state-variables.mdl` file.

```
1   State_variable ALI_sv {
2         states = ( "data", "standby", "idle", "off" );
3         transitions = ( "standby"<->"data", "idle"<->"standby",
                          "off"<->"idle");
4         default_state = "idle";
5   };
6
7   State_variable aperture_sv {
8         states = ( "open", "closed");
9         transitions = ( "open"->"closed", "closed"->"open" );
10        default_state = "closed";
11  };
```

Figure 9 State Variable Examples

Definition/Type

A state variable is defined in lines 1 and 7 of Figure 9. The definition includes the name followed by a pair of braces and a semi-colon similar to the C language syntax. Once the state variable is defined in the model file, it can be instantiated in the initial state file.

The set allowable states is defined with the “states” keyword followed by the complete list of state values that the state variable can take on, as in lines 2 and 8 of Figure 9. If no states are given, the state variable defaults to allow any state (i.e. any possible string value). The default value parameter, which must be in the state values list, is the value of the state variable at the start of the schedule. The set of possible states can also be specified with a user-defined parameter type. For example:

```
Parameter string mode { Domain = (“on”, “off”); }

State_variable mode_sv {
  states = mode;
  default_state = “on”;
}
```

In this way, a set of values can be defined in one place, and any other definition that requires those values can simply refer to the parameter type definition. This includes state variables, as above, but also reservations on that state variable, such as:

```
activity change_mode {
  mode m;
  reservations = mode_sv change_to m;
}
```

The allowable state transitions between states can be indicated using the “`transitions`“ keyword. With no transitions specified, all transitions are allowed by default. Otherwise, each legal transition is specified with a forward arrow (->) or a bi-directional arrow (<->). The ‘`all`’ keyword can be used as a short cut (e.g., `all<->”off”`). For example, in line 9 of Figure 9, the legal transitions are from open to closed or closed to open. A bi-directional arrow (<->) could also have been used to specify a transition in either direction.

State variables can also be specified as vector types. An example of vector state variable definitions and how they are used in activities is included in Figure 10.

```
State_Variable SV {
      states = <real, real, real>;
};

Activity A {
      real x;
      real y;
      real z;
      reservations = SV change_to <x,y,z>;
};
```

Figure 10 Vector State Variable Example

States are instantiated in a `.ini` (usually `timelines.ini`) file similar to activities. All states must have exactly one instance defined. Generally, no parameters are defined in the instance. However, specific instances can be assigned states or transitions that are subsets of the states and transitions in the type definition.

2.4. Parameters

The ASPEN modeling language includes parameters, which are user-defined types. Currently, only subtypes of the built-in types can be specified. Parameters can be a constant, list, or range of integers, strings, floating points, or booleans. Parameters can be defined as enumerated types for a list of states in a state variable. Ranges of values can also be used. Some examples are:

```
   parameter string ALI_mode {
      domain = (“data”, “idle”, “standby”, “off”);
   };
   parameter int WARP_range { domain = [1,40960];  };
```

In the first example, the `ALI_mode` parameter can take on any of the four values in the list. In the second example, the `WARP_range` parameter can be any integer in the indicated range from 1 to 40960. Parameters are used to pass value from activities to other activities, reservations, or functions.

The domain of a parameter can also be left unspecified. This indicates that the parameter can take on any value for the given base type. For example, a string parameter without a domain can take on any string value. An integer parameter with no specified domain can be any value between negative infinity and positive infinity (the current implementation is restricted to one half of max and one half of min integer as the bounds).

Unspecified domains can also occur in parameter declarations in activities. For example:

```
      activity A {
        string s; // any string
        int i;    // [-infinity, infinity] min and max integer values
        float f;  // [-infinity, infinity] min and max float values
        bool b;   // (true, false)
      }
```

Activity A has four parameters `s`, `i`, `f`, and `b` that can take on any string, integer, float, or boolean respectively.

2.5. Dependencies/Functions

The user can specify functional dependencies between parameters in the model. These dependencies can be either with local parameters in the activity, or with parameters in other activities. The parameters can be any of the user-defined or internal parameters (e.g. `start_time`, `end_time`, `duration`). For local dependencies, simply give the keyword "`dependencies`" followed by an equal sign and the list of dependencies. A dependency is specified by a "source" or set of source parameter names, a directional arrow (`->`, `<-` or `<->`), an optional function name, and a sink parameter name. Here's an example:

```
dependencies = p -> q, x <- f (y, z);
```

In this example there are two local dependencies. Because these are local dependencies, all parameters come from the activity for which the dependencies are defined. The first says that the value of `q` must be the same value as `p`, and that `q` will be assigned the value of `p` when `p` changes. The second says that `x` must be the result of calling the function `f` with the arguments `y` and `z`. In addition, when either `y` or `z` changes, the function `f` will be called and the result assigned to `x`. This sort of value propagation only occurs while using the Parameter Constraint Network (PCN), which is the default mode. The PCN can be disabled by running ASPEN with the `–NPCN` flag.

Dependencies can also be expressed between two parameters in two different activities. They can be between parameters of an activity and a subactivity, or between an activity and an activity required by a temporal constraint. This is done using the "`with`" expression. The parameter name on the left-hand side must refer to a local parameter and the right-hand side must refer to a parameter in the required activity. Below are two lines you might see in an activity definition:

```
decompositions = child1 with (x <-> y),
                 child2 with (d -> duration);
constraints = starts_before start of act1 with (a <- f (b));
```

The first line is a decomposition with two subactivities `child1` and `child2`. The dependency between parameter `x` within this activity and parameter `y` defined in `child1` requires that they must be equal. The dependency on `child2` says that the duration of `child2` must be equal to `d` in this activity. The third line is a constraint to the activity `act1`. The dependency in the constraint says that parameter `a` of this activity must be the result of applying function `f` to parameter `b` of `act1`.

A function in a dependency must be one of the internal functions or a user-defined C function. In order to make the C function accessible to ASPEN, the header file must be parsed by an ASPEN utility called "`apfc`" (for ASPEN Parameter Function Compiler). The parser reads in the header file (`.h`), produces an ASPEN readable source file (`.cc`), and compiles this file, along with the source file with the C functions, to produce a shared object (`.so`) library. This library can then be loaded into ASPEN at run-time with the `-F` command line flag, or from the GUI.

The signature of a parameter function is limited to set of argument and return types. Arguments and return types must be one of: `int`, `float`, `bool`, or `const char*`. These correspond to the ASPEN parameter types: `int`, `real`, `bool`, and `string`, respectively. Other than this, the function can contain arbitrary C code.

Functions for initialization and cleanup can also be defined. These functions will be called at ASPEN startup and shutdown, respectively, and must be defined exactly as below.

```
extern void initializeParameterFunctions();
extern void cleanupParameterFunctions();
```

The header file parsed by "`apfc`" must contain only the function signatures described above.

Errors

Writing parameter functions is a bit tricky. ASPEN doesn't catch errors in a very nice way (yet). The following errors could indicate a problem with a parameter function:

```
1. Warning: trying to set parameter value x to illegal value y
2. Warning: inconsistency found in PCN
   Propagating: x <- function<y, z>
```

These warnings say that a function computed a value, and that this value is not one of the possible values for the resulting parameter. So for example, if a model contains the statement, `foo <- f(bar)`, and `f(bar)` results in a value that `foo` cannot be, ASPEN will print the warning above. When writing functions like `f`, make sure that any parameter `foo`, computed with a function `f`, has a domain that includes all possible outcomes of `f` for all possible values of `bar`. Alternatively, make sure that the function `f` forces the output to be in the domain of `foo`. For example, using range parameters, this can be done with the `min` and `max` functions:

```
max(lb, min(ub, val))
```

truncates `val` off at the lower and upper bounds of the range `[lb, ub]`.

A number of internal parameter functions already exist in ASPEN. These functions can be used immediately in any model without defining additional functions. See Figure 11 for the complete list.

Function	Number of args	Arg/return type	Purpose
sum	2	Integers	Adds the two arguments
neg	1	Integers	Returns zero minus the argument
min	2	Integers	Returns the smaller of the two args
max	2	Integers	Returns the larger of the two args
mul	2	Integers	Returns the product of the two args
inv	1	Integers	Returns the inverse of the argument

Figure 11 Built-in Parameter Dependency Functions

2.6. Preference Language

ASPEN allows the user the ability to give preferences and calculate the score for ASPEN schedules. The user can add preferences with the high-level language using the `preferences.tab` file. See Figure 12 for an example of preferences file. There are five basic types of preferences:local activitiy variable, activity/goal count, resource/state variable, resource/state change count, and state duration.

Activity variable preferences rank the values of a local variable in an activity instance. Local activity variables can include start_time, end_time, duration, decomposition_index, any other domain-specific variables, or even distance from other activities (see preferences 3, in Figure 12). Activity or goal count preferences specify the optimal number of instances for a specific activity, or a generic preference for more optional goals (see preferences 2, 6 in Figure 12). Resource or state variable preferences rank the set of resource/state values that exist within the planning horizon for a specific timeline (see preferences 7 in Figure 12). Resource or state change count preferences rank the number of times a state might have changed (see preferences 4 and 5 in Figure 12). Finally, a state duration preference can rank the duration of a particular state on a state variable over the schedule horizon (see preference 8 in Figure 12).

If no preference for a parameter is specified, then it is assumed it has the maximum score (1.0). The default weight on a preference is 1, and component scores will be (sum of component scores)/(sum of component weights). The examples shown in Figure 12 are basically converted into utility functions that are used in dependencies to compute the value of "score" parameters. For example, "linear" preferences are computed as follows:

1	Prefer exponentially less conflict all total occurrences
2	Prefer exponentially more activity ali_data_take all total occurrences between 1 and 150
3	Prefer linearly more activity ali_data_take parameter start_time avg value between 400 and 3600
4	Prefer linearly less timeline solar_array avg value weighed 3
5	Prefer exponentially less timeline solar_array total occurrences
6	Prefer linearly more activity ali_moon_valibration total occurrences
7	Prefer exponentially less timeline aperture_sv max value
8	Prefer linearly more timeline sad_sv total duration of tracking value

Figure 12 Preferences.tab File Example

Preference Syntax

The preference table contains a list of preferences, one on each line. From the command line, the filename of the preference table to be loaded is given with the `–p` flag. Each line of the file should be of the form:

```
  prefer [exponentially|linearly|stepwise] [more|less|centered]
         [around <val>] [conflict|activity|timeline|parameter]
         [<type>|all] [parameter <name>] [min|max|avg|final|total]
         [duration of <val>] [value|occurrences]
         [between <val> and <val>] [weighted <val>]
```

Where everything is a keyword except for those in <> which are taken from the model.
Optional fields are "`around <val>`" (which is only for "`centered`" preferences), "`parameter <name>`" (which is only for "`activity`" parameter preferences), "`duration of <val>`", "`between <val> and <val>`", and "`weighted <val>`".

The first word on the line is simply "`prefer`" to indicate a preference. The next two words specify the type of preference function used to compute the score. The next two words are optional and specify a center value for the high score of a "`centered`" preference. The default center is the midpoint of the parameter's domain. The next word indicates where the value is coming from (a timeline, a parameter in an activity, etc.). The next word is either the type of the activity, timeline or parameter, or it indicates "`all`" activities, timelines or parameters. The next two words are only valid when the fifth word is "`activity`". They specify the name of the preferred parameter in the given activity. The next word specifies whether to use all matching values ("`total`") or to compute a function (e.g., average) on the set of matching values. The next three words are optional and only valid for timelines. They indicate a timeline value whose *duration* is preferred, rather than the value itself. The next word indicates whether the preference is on a value, or the number of occurrences of a value. The next four words are optional and specify the range of values for which the score function is computed. Values outside the range either give the minimum or maximum score. This range defaults to the domain of the parameter. The last two words specify a weight for this preference, relative to all other preferences. For example, a weight of 2 will count the score of this preference twice towards the overall score.

Types of Preferred Values

Preferences can be made on four difference types of values: the number of conflicts in the schedule, the number of activities in the schedule, the value of a timeline, or the value of an activity parameter. To indicate a preference on the number of conflicts, the user must specify the keywords "`conflict all total occurrences`." No other combinations with "`conflict`" are allowed. Also, "`duration of`" is not allowed. All other fields are valid. (Although "`more`" and "`centered`" preferences are allowed, "`less`" is obviously most common.) The default "`between`" range is from 0 to infinity. A typical conflict preference might look like:

```
      prefer exponentially less conflict all total occurrences
```

For preferences on the number of activity instances, the user must specify "`activity <name> total occurrences`" where <name> is the preferred activity type name. In this way, the user can give higher scores to schedules with more or less activity instances of a given type. The user can also specify the preference for a particular number of instances using the "`centered`" and the optional "`around`" keywords. The default "`between`" range is from `0` to infinity. A typical activity preference might look like:

```
      prefer exponentially more activity data_take total occurrences
```

Preferences can also be made on the values on timelines. A timeline preference can be on: the minimum, maximum, average, or final; value, duration of a value, or occurrences of a value on the timeline. The number of occurrences on a timeline is simply the number of times the resource or state variable changes value over time. The default range is from 1 to infinity (there is at least one value on any given timeline). Typical timeline preferences might look like:

```
      prefer linearly less timeline power max value
      prefer linearly less timeline camera_switch avg duration of
             on value
      prefer exponentially less timeline power total occurrences
```

Finally, preferences can be made on the values of parameters in activities. They can be made on all parameters of a given type using only the "`parameter`" keyword. Or they can be made on all parameters with a given name in a given activity type. This is done by giving first the "`activity`" keyword then the "`parameter`" keyword. The default range is the same as the range of values for the parameter. Typical parameter preferences might look like:

```
      prefer exponentially centered parameter temp avg value
      prefer linearly centered around 0 activity picture parameter
             angle_off_target avg value
```

Preference Functions

Preferences define a function between values in the schedule to a score representing the schedule quality. There are three types of preference functions: linear, exponential, and stepwise. For each type, the user can specify the direction of the function: more, less or centered.

Linear preferences allow the user to specify a score that increases linearly with the value of the parameter. The score makes a straight line from the endpoints of the score and parameter values.

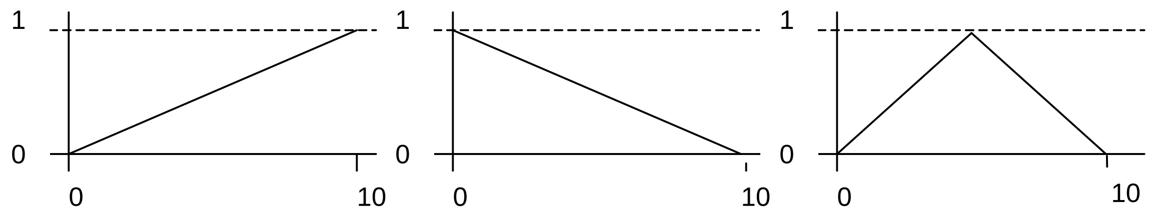


Figure 13 Linear preference functions for a) increasing, b) decreasing and c) centered

For example, a “`more`” preference can be made on a parameter p that has a range from 0 to 10. If no preference range were given, the score would be 0 when p was 0 and 1 when p was 10 (see Figure 13).

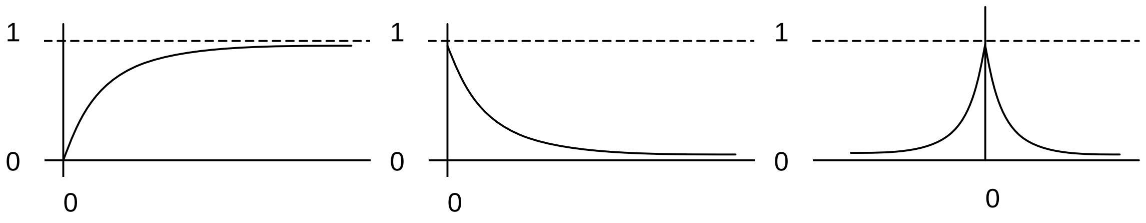


Figure 14 Exponential Preference functions for a) increasing, b) decreasing, and c) centered

Exponential preferences allow the user to specify a preference function on parameters with possibly infinite domains. The score value changes exponentially with the value of the parameter. For example, a “more” preference can be made on a parameter that has an integer domain from 0 to infinity. The score will approach the maximum score as the value of the parameter approaches infinity (see Figure 15).

Stepwise preference functions are for parameters with discrete domains i.e., a fixed set of values. To calculate the incremental step in score, the score is divided by the number of values in the parameter domain. For example, a “more” preference can be made on a parameter p that can be any value in the set (a, b, c, d, e). The possible values of p are ordered, and the score increases with each value (see Figure 15).

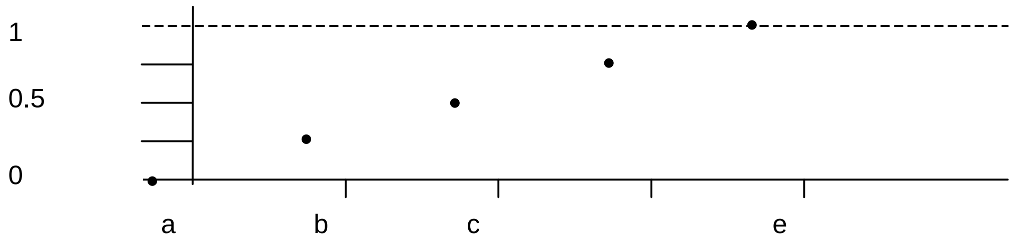


Figure 15 Stepwise preference function

The user can also specify a range for the score of the preference with the optional “`between`” keyword. The range represents cutoff points where the score either hits 0 or 1 (the minimum and maximum score values). In the linear example above, if a preference range of “between 2 and 8” were given, the score would be 0 when p was 2 or less and 1 when p was 8 or more. See Figure 16 for an example of “between.”

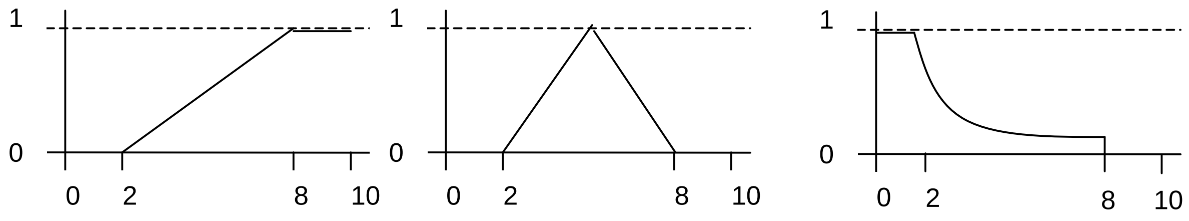


Figure 16 Examples of preferences with "between" ranges for a) linearly more, b) linearly centered and c) exponentially less

For centered preferences, the user can give a value for which the preference is centered around. If not specified, the default center is the midpoint of the domain of the parameter. The center value skews the function to one side or the other.

Each preference counts towards the total score for the plan/schedule. The total score is calculated by multiplying each individual preference score by its weight, summing this value, and dividing by the sum of the weights.

2.7. Generating Output Files

ASPEN allows the user to generate several different types of output files. With the exception of command output (described below) all of them can be saved by simply using the `–o` command line argument or from the GUI (under the `File` menu). The entire schedule can be saved in the `.ini` format that is readable by ASPEN. This saves all timeline and activity instances currently in the schedule. Resetting ASPEN and loading this saved file will return ASPEN back to the exact state at which it was saved. Alternatively, formatted commands can be written to a file for each activity instance in the schedule. Formatting these commands is described below. Finally, various parts of ASPEN can be exported to different types of files for viewing and debugging the model and schedule. The TCN and PCN can be exported to a file readable by a network visualization program called `xvcg`, which displays a graphical view of the nodes and edges (for X windows only). The model (including activity, timeline, and parameter types) can be exported to HTML, which can be viewed by a web browser such as Netscape. Using HTML, the activities, timelines, and parameters are hyper-linked to other activities, timelines and parameters for easy cross-referencing.

Command Output

In the spacecraft domain, mission operations personnel are interested in a time ordered sequence of commands as the output of the planning system. In ASPEN, the structure of the command output file is specified in the “command format” file. Each line of the file denotes an entry of the file output format of an activity type. An entry begins with the name of the activity type followed by a colon (:). Then the format of the output for that activity type is specified. The white spaces between the colon and the first non-white space character are ignored. A parameter is referenced by specifying `$(parameter_name)` in the file. A new line character is represented by $n. Any other characters are printed verbatim in the output. For example:

```
      as1: com1 $(start_time) $(end_time) $(param2),$(param1),$(param3)
      as2: com2 $(start_Time) $(end_time) $(param1)
```

The EO-1 model uses the following command format file:

```
      ali_data_take: $(start_time) alidt
      ali_changer: $(start_time) ali_mode $(ali1)
      sad_changer: $(start_time) sad_mode $(sad1)
      aperture_changer: $(start_time) aperture_mode $(aper1)
      engdata_changer: $(start_time) eng_data_mode $(eng1)
      acs_changer: $(start_time) acs_mode $(acs1)
      ali_sun_calibration: $(start_time) ali_suncal
      slew_to_sun: $(start_time) sunslew
      ali_moon_calibration: $(start_time) ali_mooncal
      slew_to_moon: $(start_time) moonslew
      roll_to_next_position: $(start_time) roll
      maneuver: $(start_time) maneuver
      # comments start with the number sign
      thruster_warmup: $(start_time) thruster_heater_on
      ephemeris_upload: $(start_time) ephem_upload
      command_upload: $(start_time) command_upload
      clock_data_upload: $(start_time) clock_data_upload
      ephemeris_validate: $(start_time) ephem_validate
      command_validate: $(start_time) command_validate
      downlink: $(start_time) downlink $(amount)
```

The following is a partial listing of the command file output from the EO-1 planning process:

```
      1998-148/00:00:00 moonslew
      1998-148/00:01:39 sad_mode "fixed"
      1998-148/00:02:35 aperture_mode "open"
      1998-148/00:02:39 ali_mode "standby"
      1998-148/00:05:39 eng_data_mode "high"
      1998-148/00:06:09 acs_mode "science"
      1998-148/00:06:39 ali_mode "data"
      1998-148/00:06:40 alidt
      1998-148/00:07:05 ali_mode "standby"
      1998-148/00:07:07 aperture_mode "closed"
      1998-148/00:07:07 sunslew
      1998-148/00:07:20 sad_mode "track"
      1998-148/00:07:24 ali_mode "data"
      1998-148/00:07:25 ali_mode "standby"
      1998-148/00:07:26 ali_mode "idle"
      1998-148/00:07:35 acs_mode "nadir"
      1998-148/00:08:05 eng_data_mode "low"
      1998-148/00:08:29 ali_mode "standby"
      1998-148/00:12:30 ali_mode "data"
      1998-148/00:12:30 ali_suncal
      1998-148/00:12:31 ali_mode "standby"
      1998-148/00:12:50 ali_mode "data"
      1998-148/00:12:51 ali_mode "standby"
```

The command format file name is specified at start up using the `–C <filename>` option.

3. Running ASPEN

ASPEN can be run in three different ways: batch mode, with the GUI, or interactive mode. For all methods, a `setup.csh` script at the top-level of the ASPEN directory needs to be sourced, in order to set the correct environment variables. ASPEN is then invoked from the command line using the necessary parameters that are listed in Table 6. Most built models in ASPEN include scripts that will start ASPEN with the correct parameters. These scripts are usually located in the `/bin` directory. The scripts take one argument, which switches as below:

`j` - run with the Java GUI
s - run aspen as a server (no GUI)
b - run aspen in batch mode (no GUI, plans, saves, and exits)

If you give no argument, aspen will run with the command shell user interface.

-h		Prints out all the parameters, either *h* or *help* can be used
-V		Show the version information
-g		Runs ASPEN with the X GUI (must have been built)
-S	*port (optional)*	Runs ASPEN as a server listening to a socket with the given port number or with any available port
-j		Runs with the Java GUI client (requires -S)
-m	filenames	Specify the model file names after this parameter
-i	filenames	Specify the ini filenames after this parameter
-f	function	Specify the scheduling function to use
-b		Run in batch mode
-c	filename	Use the specified canned plan file
-s	seed	Use the specified random seed
-r	iterations	When running in batch mode, repair the schedule with the given maximum number of iterations.
-o	filename	Specify the output filename
-F	filename	Specify the external functions library for the model
-h	filename	Specify the filename for the heuristics table
-H	filename	Specify the heuristic functions library for the table
-C	filename	Specify the file that describes the output format for executable commands
-p	filename	Specify the filename for the preferences table
-U	filename	Specify the user-defined scheduling functions library
-NTCN		Run without the temporal constraint network
-NPCN		Run without the parameter constraint network

Table 6 ASPEN Command Line Parameters

An example invocation of ASPEN is the following:

```
 % cd models/test
 % aspen -S -j -F functions.so –h heuristics.tab -m test.mdl –i test.ini
```

This example would run ASPEN as a server with the Java GUI, using the specified external functions library, heuristics table, specified model file, and initial state file. If the random seed is not specified, ASPEN will generate a new random seed when it is invoked. Similarly, if an output filename is not specified then a default filename is used, and the output is written to the top-level output directory.

3.1. Batch mode

When running in batch mode, any output to standard out or standard error can be piped to a file. The `-g`, `-j`, and `-S` options are not valid when running batch. If the `-r` option is used, ASPEN will run the repair algorithm and attempt to generate a conflict-free schedule.

3.2. Interactive mode

Running in interactive mode starts up ASPEN in a shell command loop. This allows the user to look at details of the schedule with a text interface. Also, activities can be created, moved, and deleted in this mode. Typically, the user enters interactive mode by specifying the model and initial state files (plus the external function library and the heuristics table) which are loaded into ASPEN. An example is:

```
  % aspen -F functions.so -h heuristics.tab –m test.mdl -i test.ini
```

A number of commands can be used in interactive mode. See Table 7.

abstract <name>	abstracts (removes) the subactivities of activity *name*
change <act> <param> <value>	changes the parameter *param* of activity *act* to *value*
create <name> <time>	creates an activity of type *name* at *time*
delete <name>	deletes the activity *name*
detail <name>	details (adds) the subactivities of activity *name*
dispatch	runs the forward dispatch scheduling function
display [sdb|schema|timeline(s)|activity(ies)] <name>	displays the current contents of the given object
duration <name> <integer>	changes the duration of *name* to *integer*
generate <n>	randomly generates conflicts by creating or moving *n* random activities
help	lists available interactive commands
log	starts logging operations (for *undo*)
move <name> <time>	moves the activity *name* to *time*
nlog	stops logging operations
place <name>	places the activity *name* on the timelines
quit	quit aspen. Queries if you want to save the schedule
remove <name>	removes the activity *name* from the timelines
repair <n>	runs the iterative repair scheduler
reset	resets the schedule data-base
save [ini|scr|pcn|tcn] <filename>	saves the schedule in the specified format
simple	runs the simple placement scheduling function
undo	undoes the last operation

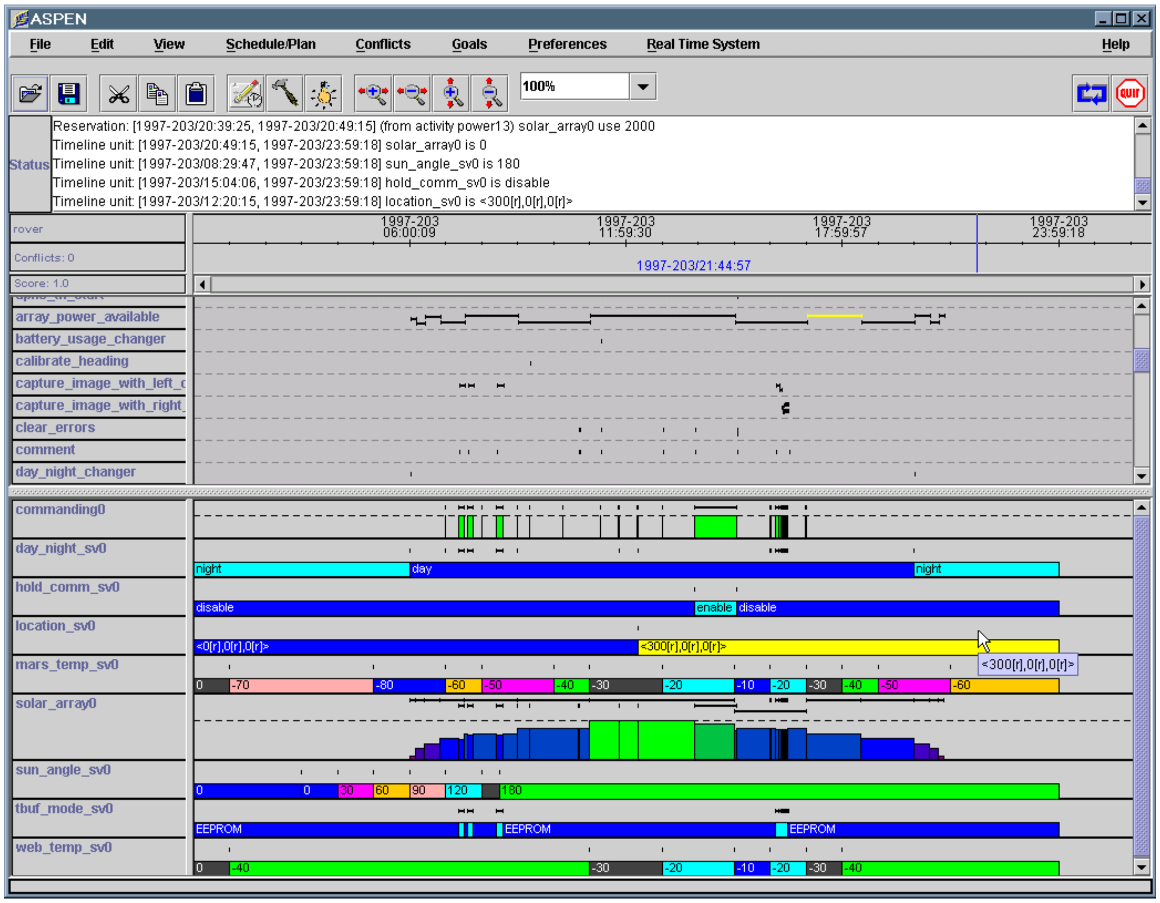
Table 7 Interactive Mode Commands

Generally, the user would then call "`dispatch`" to do an initial placement of the activities, and if necessary, then run repair with a specified number of iterations to get a conflict-free schedule. Then, particular timelines or activities can be displayed. Activities can be moved, created, or deleted. There is also a “`generate`“ option that randomly creates or moves activities to deliberately try to create conflicts in the schedule. This is for testing purposes.

3.3. Graphical User Interface (GUI)

The ASPEN Graphical User Interface (GUI) component provides tools for graphically displaying and manipulating schedules. Resource and state timelines are displayed. Activities are overlayed on the timelines, and users can directly manipulate activities using standard drag-and-drop procedures. The typical invocation is:

```
% aspen -j -S -F functions.so -h heuristics.tab -m test.mdl –i test.ini
```



```
Reservation: [1997-203/20:39:25, 1997-203/20:49:15] (from activity power13) solar_array0 use 2000
Timeline unit: [1997-203/20:49:15, 1997-203/23:59:18] solar_array0 is 0
Timeline unit: [1997-203/08:29:47, 1997-203/23:59:18] sun_angle_sv0 is 180
Timeline unit: [1997-203/15:04:06, 1997-203/23:59:18] hold_comm_sv0 is disable
Timeline unit: [1997-203/12:20:15, 1997-203/23:59:18] location_sv0 is <300[r],0[r],0[r]>
```

Figure 17 ASPEN G.U.I.

This will load in the specified model and initial state files. Then, in the GUI, the user hits the ‘*Plan*’ button to run the forward dispatch scheduling algorithm. Once this completes (which is shown in the status window), timelines will display resource and state values throughout the horizon. If there are any conflicts, they are shown in red on the timelines and listed under the ‘*Conflicts*’ pulldown menu. In addition, a count of the number of conflicts can be seen in the upper left hand corner of the screen under the ‘*Load a file*’ button. Conflicts can be resolved, by hitting the ‘*Repair*’ button (looks like a hammer). This will run the iterative repair algorithm for a default number of iterations. When the repair algorithm completes a message is displayed in the status window. If not all conflicts have been resolved, then the ‘*Repair*’ button can be hit again.

The schedule can be zoomed in and out, both vertically and horizontally, using the '*Zoom*' buttons. The GUI view can also be changed to one minute, hour, day, week, or a percentage of the horizon length using the sizing pulldown (default is 100%). Like the interactive mode, activities can be created (added), moved, deleted, detailed, abstracted, placed, and lifted. To create an activity, pull down the '*Edit*' menu, and choose '*Create activity*'. A menu will pop up which will allow the user to pick an activity to add. Then simply click on the activity timeline at the location where that activity should be placed in the schedule. Similarly, moving and deleting activities is accomplished simply by choosing '*Move activity*' or '*Delete activity*' from the '*Edit*' menu, and then clicking and highlighting the given activity in the schedule. All actions of adding, moving, deleting activities can be logged, by choosing '*Set Logging On*' from the Timeline menu. Once logging is on, any action can be undone by clicking on the '*Undo*' button. The logger keeps track of up to 100 actions. The entire schedule can be displayed to a window and saved (to be inputted as a 'canned plan' in the future). The schedule can be reset, just as in the interactive mode, which then allows the user to reload and schedule model and initial state files.

The Java GUI can also be run on a PC. The Java Runtime Environment (JRE) must be copied and installed from the Sun website (java.sun.com). The user must then copy the ASPEN Java GUI libraries (`*-classes.zip`) on to the PC. The user will then be able to run the GUI on the PC with the server running at a Sun workstation. The speed of this setup will depend on the network connection between the PC and the Sun.

The following tables describe each of the menu options included in ASPEN.

File Menu	**Description**
Load New Project	Used to load ASPEN model files (.mdl)
Load Instances	Used to load ASPEN instance files (.ini)
Save	
Save As ⇒	Save Schedule as… outputs an .ini file of the activities in the current schedule Save Commands as… outputs an ascii list of spacecraft commands corresponding to the activities in the current schedule. The command to activity mapping is defined in the commands.fmt file.
Export	
Export As ⇒	Export HTML will save an html representation of the schedule. Export PCN will save the parameter constraint network file (for display with "xvcg" on X windows) Export TCN will save the temporal constraint network file (for display with "xvcg" on X windows)
Refresh	Refreshes the GUI.
Reset	Resets ASPEN with an empty schedule (and no definitions)
Quit	Quits ASPEN and the GUI.

Edit Menu	Description
Select an Activity	Examine the parameters for a single activity selected from the complete list. The list contains the start times of the instances.
Create Activity	Create a new activity. After selecting this menu option, a new menu will appear listing activity types. After selecting an activity, the user will place the mouse on the activity portion of the GUI and press the right mouse button. This will add the activity at that point in the timeline.
Move Activity	Move an existing activity. This will move the selected activity if there is one (activities are selected by single-clicking on them with the left mouse button). Otherwise, it will pop-up a list of activity instances that can be moved. The user can also move activities with the mouse using drag and drop.
Delete Activity	Delete an existing activity. This will delete the selected activity if there is one. Otherwise, an activity can be selected from a new window that will appear listing the activity instances.
Detail Activity	Detail an existing activity (i.e. create sub-activities). This will detail the selected activity if there is one. Otherwise, an activity can be selected from a new window that will appear.
Abstract Activity	Abstract an existing activity (i.e. delete sub-activities). This will abstract the selected activity if there is one. Otherwise, an activity can be selected from a new window that will appear.
Place Activity	Place an existing activity on the timelines. This will make any reservations the activity has on resources and state variables. The corresponding timelines will reflect these usages.
Remove Activity	Remove an existing activity from the timelines. This has the reverse effect of placing. All reservations for this activity will be removed from the resources and state variables.
Options	Change the global behavior of schedule edits, including applying edit to sub-activities, applying edit to connected activities, automatically scheduling activities when they are created, and ignoring the permissions when editing activities.
Enable Logging	Starts a log of scheduling operations (e.g. move, add, delete). Once logging is on, any action can be undone by clicking on the ‘Undo’ button.
Enable Auto-Detail	Change the edit mode to automatically detail activities (i.e. create sub-activities) when they are created. Other edit operations will also be performed on sub-activities. For example, a move with auto-detail will move sub-activities along with the activity being moved.
Enable Auto-Connect	Change the edit mode to automatically connect temporal constraints, creating activities when needed. Other edit operations will also be performed on connected activities. For example, a delete with auto-connect will disconnect and delete connected activities.
Enable Auto-Schedule	Change the edit mode to automatically move subactivities and/or connected activities to legal locations when they are created by detail or connect operations.
Use Permissions	Change the edit mode to either use or ignore the permissions specified for activities. When permissions are ignored, any edit operation is valid.

View Menu	Description
Fit to Horizon	Changes schedule view to horizon length defined in model.mdl file.
Fit to a Minute	Changes schedule view to one minute.
Fit to an Hour	Changes schedule view to one hour.
Fit to a Day	Changes schedule view to one day.
Fit to a Week	Changes schedule view to one week.
Smart Fit	Changes schedule view to end at time of last scheduled activity.
Horizontal Zoom In	Decreases schedule view horizontally.
Horizontal Zoom Out	Increases schedule view horizontally.
Vertical Zoom In	Decreases schedule view vertically.
Vertical Zoom Out	Increases schedule view vertically.

Timelines Menu	Description
Display/Hide Timelines	Add or remove activites and resource and state timelines from the GUI view. The new view can be saved and reloaded.
Display/Hide Status Window	Add or remove the status window from the GUI.
Display/Hide Empty Timelines	Add or remove timelines with no reservations.
Display/Hide Activity Name	Add or remove activity names from appearing beside the activity rows.
Display/Hide Reservations	Add or remove reservation timelines from appearing above each resource and state timeline.
Zoom	Horizontal or vertical zoom in or out.
Pan/Zoom to location/ duration	Zooms to a given start and end time.
Smart fit	Fits display to the specified duration.

Schedule/ Plan Menu	Description
Simple Placement	Place the current set of activities on the timelines.
Forward Dispatch	Schedule algorithm used for initial schedule generation.
Timeline Manager	Similar to forward dispatch, except it will remove activities that are found to violate timeline conflicts. The choice of which activity to remove is based on preemption, which uses the priority of an activity to make the decision.
Pack Activities	Algorithm which pushes each scheduled activity to its earliest possible start time
Iterative Repair	Local search algorithm used for solving schedule conflicts.
Iterative Optimize	Local search algorithm used for optimizing user preferences.
User-Defined Schedule	This runs the user-defined scheduling function is one was specified (see the section on “Procedural Scheduling”).

Conflicts Menu	Description
{list of conflicts}	See Table 8 for conflict type descriptions. Selecting conflict will change GUI view start time to conflict start time at the current plan magnification.

Goals Menu	Description
Satisfied	All of the goals which were declared in ASPEN .ini files and have been placed in the current schedule will appear in the satisfied pulldown list.
Unsatisfied	All of the goals which were declared in ASPEN .ini files and have not been placed in the current schedule will appear in the unsatisfied pulldown list.

Preferences Menu	Description
{list of preferences}	All of the preferences specified in the model's preferences.tab file will be listed here with a checkbox indicating whether that preference is being used in the optimization and scoring.

Real Time System Menu	Description
	This function is not yet documented.

Any options, including a file specifying the timelines and activity windows to be displayed, can be saved out to an options file and will automatically reload when that model is run again with the GUI. The GUI should be run in its own model file. The options file will allow the user to specify preferences such related to the behavior of the repair and optimization algorithms, the presence of the status window, empty timelines, activity names and reservations, the font size within the GUI, the size of the GUI window, and the number of iterations (or maximum time) for the performance of repair or optimize (among others). The current default for any options error is the original specifications for the GUI.

3.4. *Building an Initial Schedule*

Typically, different algorithms are used to generate an initial schedule than those used to repair or modify existing schedules. In ASPEN, we have implemented one such algorithm called Forward Dispatch. Forward Dispatch first details and connects the constraints of all activity instances. This involves creating many new activities. Then, it sorts the new set of activities by increasing latest start time. In this order, a legal start time is computed for (and assigned to) the activity. Legal time intervals for an activity are calculated by intersecting legal intervals for its resource and state timelines with the legal intervals defined by the activity's temporal constraints. If no legal times were found, it will ground the activity at its earliest start time. With a grounded start time, the activity can then be placed on the timelines. This initial schedule generator is not guaranteed to produce a conflict-free schedule.

3.5. *Repairing Conflicts in a Schedule*

If there are conflicts indicated in a given schedule after running the scheduling algorithm (e.g. forward dispatch or simple placement), then we attempt to resolve the conflicts with the repair function. To execute the repair function, either hit the repair button on the GUI, or execute the repair command in the interactive mode. Repair can also be automatically be run in batch mode with the '`-r`' option. Repair then attempts to resolve each conflict individually, for the given number of iterations (default is 200 in the GUI) or until it finds a conflict-free schedule. Some basic statistics are printed out at the end of execution telling how many conflicts there were to begin with, how many there were at the end of repair execution, what percentage of the conflicts were solved, and how long it took to resolve the conflicts.

Conflict Type	Description
Depletable Resource	Resource that is diminished after use (*eg*, fuel). Conflict occurs when resource is over- or under-subscribed.
Mismatched Decomposition	A decomposition index value does not match the actual decomposition of an activity.
Non Depletable Resource	Resource that is not diminished after use (*eg* solar array power). Conflict occurs when resource is over- or under-subscribed.
Open Constraint	Conflict occurs when an activity *a* needs to connect to an activity *b*, but has not connected to it.
State Usage	Conflict occurs when a state timeline unit is in a state *a*, but the activity on that unit requires a state *b.*
State Transition	Conflict occurs when a pair of state timeline units are not a legal transition.

Temporal	Conflict occurs when two connected activities' constraint distance is violated.
Undetailed Activity	An abstract activity which needs to be detailed.
Unground Parameter	Parameters can be declared as being a range [,] or a list (,). Until a single value is chosen for these parameters, they cause a conflict. Note: start time, end time, and duration are also parameters (as well as user-defined local variables).
Unplaced Activity	A defined and created activity which has not been assigned and placed on a timeline.
Unsatisfied Goal	A mandatory goal which the user specified in a model file has not been placed in the schedule.
Violated Dependency	Parameters can be dependent on one another through a given function. Conflict occurs when this dependency function is violated.

Table 8 Conflict Types

Conflicts in a given schedule are resolved through the iterative repair function. The repair function attacks each conflict individually, applying the appropriate function in an attempt to resolve the conflict.

For the resource conflicts, repair attempts to find an activity to add or move in order to resolve the conflict on that timeline unit. For example, if two activities are using a resource that can only be used by one activity at a time, then repair will attempt to move one of the two activities to another part of the schedule. Or, if a resource has been over-subscribed (such as battery power), then it might find an activity to add which would add more of the resource to eliminate the over-subscription conflict. As a last resort, repair will try to delete activities. The behavior of repair can be managed by writing heuristics that help direct how repair will try to resolve certain types of conflicts. (See the Heuristics section for more details.) Also, repair checks to see what permissions a given activity has before attempting to move, add, or delete.

For open-constraint conflicts, the repair function either attempts to move an existing activity and connect to it, or add a new activity and connect to it. An example of an open-constraint conflict would be when an activity *a* should connect to an activity *b.* If *b* is already in the schedule and can be moved, then *a* will connect to *b*. Otherwise, repair will create activity *b* and then place it on the timeline, connecting it to activity *a*.

State usage conflicts occur when a user on a state timeline requires a state that is different than the value of that timeline unit. For example, a *door* activity might require that the state of door be *open*, but the state of the door is actually *closed*. To resolve this conflict, either the *door* activity is moved to a part of the timeline where the state is *open*, or a state-changer activity can be added which will cause the door to be *open*. Again, the *door* activity could simply be deleted which would also resolve this conflict, but this is a last resort operation that can be controlled by the user (either through permissions or heuristics).

A state transition conflict occurs when a pair of state timeline units is not a legal transition. For example, a *door* activity can be defined in the model as having states: *open, partway,* and *closed*, with legal transitions only being: *open<->partway, partway<->closed*. So, an illegal transition would be *open->closed*. If two state timeline units have values of *open* and *closed*, respectively, then this would be a state transition conflict. To solve this conflict, the repair function would try to move one of the two state changers for those two timeline units, or try to add another state changer to make the transition legal. In the example of *open->closed*, a state changer to the state *partway* could be added between these two timeline units which would resolve this conflict.

Temporal conflicts occur when the distance constraint between two activities is violated. For example, an activity *a* is connected to activity *b* by a distance range of [0,5], but activity *b* is 10 time units away from activity *a*. To resolve the conflict, repair tries to move either *a* or *b*, to make the distance between the two activities legal.

Undetailed activity conflicts are simply resolved by detailing the activity. Detailing will add and place the sub-activities for the given activity.

Parameters can be a range or list of values, and are conflicts if they are not ground to a single value. Start time, end time, and duration of activities are all parameters too, and must be ground to a single value. Choosing a valid value for the parameter repairs an unground parameter conflict. This value may be chosen at random, or it can be determined through user-defined heuristics.

Unplaced activity conflict is resolved simply by placing it on a timeline. The activity has been created in the schedule database, but is a conflict because it has not been placed on a timeline.

Parameters can be dependent on each other through a dependency function. A violated dependency conflict occurs when this function has been violated. To fix this conflict, repair simply applies the dependency function.

Heuristics

While resolving a conflict, the repair function must make choices at each step in the process. These steps where choices can be made are called *choice-points.* At present, the *choice-points* are: choosing a conflict, choosing a conflict resolution method, choosing an activity to move, choosing an activity (schema) to add, choosing an activity to delete, choosing an open constraint leaf, choosing a valid interval (to place an activity), choosing a start time for an activity, choosing a duration for an activity, choosing a parameter (and value) to ground, and choosing a decomposition to detail in an activity. At each *choice-point*, a set of heuristics can be written which will manage how that choice will be made. For example, a heuristic can be written which will say that when trying to resolve a temporal conflict, then we should always try the MOVE resolution method to resolve it.

There are currently several heuristic functions already implemented in ASPEN. These functions are in the source code directory in a file called `heuristic-functions.cc`. (Additional heuristics can be added to this file, but this requires a few additional changes in the code -- initializing a table in `main.cc`, basically, so that this should only be undertaken with the aid of one of the primary ASPEN developers.) The existing heuristic functions (and whether they can be used for repair, optimizing, or both) are:

- `LeastMvmtIntervalHeuristic (repair)`
- `LeastMvmtStartTimeHeuristic (repair)`
- `startTimeIntervalResolveHeuristic (repair)`
- `startTimeIntervalHeuristic (repair)`
- `startTimeHeuristic (repair)`
- `randomStartTimeHeuristic (both)`
- `middleStartTimeHeuristic (repair)`
- `durationHeuristic (repair)`
- `leastChgDurationHeuristic (repair)`
- `movableActivityHeuristic (repair)`
- `activityPriorityHeuristic (repair)`
- `activitySchemaPriorityHeuristic (repair)`
- `conflictHeuristic (repair)`
- `minConflictHeuristic (repair)`
- `fewerConflictsHeuristic (repair)`
- `defaultParameterValueHeuristic (repair)`

- `resolutionMethodHeuristic (repair)`
- `abstractActivityHeuristic (both)`
- `cancelReservationHeuristic (both)`
- `liftActivityHeuristic (both)`
- `disconnectConstraintHeuristic (both)`
- `optimizationMethodHeuristic (optimize)`
- `preferenceHeuristic (optimize)`
- `conflictPreferenceHeuristic (optimize)`
- `conflictGoalPreferenceHeuristic (optimize)`
- `startTimeIntervalOptimizeHeuristic (optimize)`
- `satisfyGoalHeuristic (both)`

To use the existing heuristics, the user must specify the heuristics in a `heuristics.tab` file in the model directory. There should already be a sample `heuristics.tab` file in the existing model directories. An example from the EO-1 model is:

Choice-Point	**Heuristic Fnc Name**	**Confidence Level**	**Branching Factor**
conflict	conflictHeuristic	99	100%
resolutionMethod	resolutionMethodHeuristic	100	100%
validInterval	leastMvmtIntervalHeuristic	100	100%
startTime	startTimeHeuristic	99	100%
startTime	randomStartTimeHeuristic	1	100%
duration	durationHeuristic	100	100%

The first column is the *choice-point* name. The list of names is in the source directory in the file `aspen-common.h`. The full list of *choice-points* is: `conflict`, `resolutionMethod`, `activityToMove`, `activityToAdd`, `activityToDelete`, `activityToConnect`, `activityToAbstract`, `activityToLift`, `activityToChangeDuration`, `activityToChangeParameter`, `reservationToCancel`, `constraint`, `openLeaf`, `validInterval`, `startTime`, `duration`, `parameter`, `groundParameter`, `decomposition, and timeLine`. The choice-point name is case insensitive. The second column is the name of the heuristic function. The full list of pre-defined heuristics is given above. User-defined heuristic function names would go here as well. The third column is the confidence level. Several heuristic functions can be specified for a given *choice-point*. The confidence levels for the heuristic functions for that *choice-point*, should be less than or equal to 100. If the total is less than 100, then ASPEN will fill in with a random heuristic which choose values randomly. If the total is over 100, then the table is normalized to 100. Lastly, the fourth column indicates the branching factor. This indicates that at each choice, how many choices will be evaluated. If the branching factor is 100%, then all possible choices are evaluated. If the branching factor is, say, 50%, then only half of the choices will be evaluated.

The user can change the values of the confidence levels and branching factors in the `heuristics.tab` file. A recompile is not necessary in order to re-run ASPEN with the new values.

Additionally, the user can specify his/her own heuristics in the model directory. Writing these functions requires a more extensive knowledge of ASPEN. The functions must have the following signature:

```
     extern bool functionName(list<T>& outList, ScheduleDB* sdb,
                              Repairer* rs);
```

where `T` is the type determined by the choice point and `outList` is the resulting list of values selected by the heuristic function. The `ScheduleDB` and `Repairer` are ASPEN objects that are used to access information from the ASPEN scheduling database and the current state of repair.

The header file for the heuristic functions must be compiled with the ASPEN utility "`ahfc`" (for ASPEN Heuristic Function Compiler). Using the template `Makefile` and running '`make`' in the model directory should parse and compile the heuristic-functions into ASPEN. (This has not been completely tested yet, though.) The user can also define initialize and cleanup functions for the heuristics. These functions will be called at ASPEN startup and shutdown, respectively. The function headers must look exactly like:

```
     extern void initializeHeuristicFunctions();
     extern void cleanupHeuristicFunctions();
```

When invoking ASPEN, then a '`-H <heuristic-function-filename>`' must be added to the command line to access these functions.

Iterative Optimization

If the current score of a model is less than 1.0, then we can attempt to optimize the schedule (and improve the score) with the optimize function. To execute the optimize function, either hit the optimize button on the GUI, or execute the optimize command in the interactive mode. Optimize can also be automatically be run in batch mode with the '`-O`' option. Optimize attempts to improve each preference individually, for the given number of iterations (default is 200 in the GUI) or until it finds an optimal schedule (with a score of 1.0). Some basic statistics are printed out at the end of execution telling the initial score, the final score, what percentage improvement, and how long it took to perform these optimizations.

Optimization works similarly to repair in that it chooses a sub-optimal user-specified preference and attempts to improve the local score. The algorithm does not attempt to optimize the score, or perform hill climbing so that the overall score is guaranteed to improve at each step. In fact, the schedule might contain new conflicts when the iterative optimization is run. Optimization will run for a given number of cycles (default is 200) and if the score is 1.0 or the iterations complete, then optimization is finished. If one iteration of optimization yields a conflicted schedule, then the algorithm will perform iterative repair on the schedule until the schedule is conflict-free before continuing to optimize preferences.

The heuristics that iterative optimization uses to perform the optimization are specified and executed similarly to the repair heuristics. The heuristic is specified in the `heuristics.tab` file along with the repair heuristics, with the word 'optimize' in the first column. The choice points for optimization heuristics are: `activityToAdd, activityToDelete, activityToConnect, activityToAbstract, activityToLift, activityToChangeDuration, activityToChangeParameter, reservationToCancel, startTime, duration, parameter, groundParameter, decomposition, timeLine, preference, optimizationMethod, and satisfyGoal`. The heuristics can be specified exactly as with the repair heuristics other than ensuring that the keyword "optimize" is in the first column.

When the iterative repair algorithm reaches a choice point in the iterative optimization algorithm, it will stochastically from the set of optimization heuristic functions for the current choice point specified in the `heuristics.tab` file. The algorithm will execute this function in order to determine the optimization activity to perform and the appropriate parameter values to choose.

Optimization heuristics can be written in the same `heuristic-functions.cc` file that the repair heuristics functions are written in. The header for optimize functions is slightly different:

```
extern bool functionName(list<T>& outList, ScheduleDB* sdb,
                              Optimizer* rs);
```

The `Optimizer` type implies that the heuristic function can be used for optimization only.

```
extern bool functionName(list<T>& outList, ScheduleDB* sdb,
                                    Searcher* rs);
```

The `Searcher` type implies that the heuristic function can be used for either repair or optimization.

The user has the option of saving out the highest-scoring feasible (i.e., no conflicts) schedule each time it is achieved during iterative optimization, so that the best schedule can be reloaded at the end of the process. This option can be set in the options file under `Schedule/plan -> Reload best plan.`

3.6. User-Defined Procedural Scheduling

ASPEN comes with a set of generic scheduling functions that are useful for many scheduling applications. However, for more difficult or very specific applications, the user can write his/her own functions for building schedules of activities. Writing these functions requires knowledge of the C++ programming language as well as the object-oriented interface to ASPEN. The functions are written in a separate file, compiled as a shared library, and linked into ASPEN at run time (with the `–U` flag).

3.7. Error Messages/Output Messages

Loading Model

Table 9 lists the error messages that can occur when loading the model at start up. The description for each error message is included.

Warning	**Description**
adding activity failed ...	tried to add an activity of a given type, but something went wrong and the activity was not added
called connectConstraint on already connected constraint	tried to connect a constraint (designate an activity to satisfy the temporal constraint) but the constraint already had an activity
calling addActivity on an existing activity	tried to add to the data-base, an activity instance that was already present
calling disconnectConstraint on unconnected constraint	tried to disconnect a constraint that does not have a designated activity to begin with
calling removeActivity on a nonexisting activity	tried to remove an activity from the data-base that that is not present
cannot delete a subactivity without deleting parent!	tried to delete an activity that is a subactivity of a decomposition of another activity; this can only be done with “abstract”
cannot delete activity without deleting subactivities!	tried to delete an activity that has instantiated subactivities; must abstract this activity first
change duration called with same time	no-op
connecting constraint failed	tried to connect a constraint, but something went wrong and it was not connected
did not find activity named ...	when moving, deleting, connecting, etc., an activity instance name was specified that does not exist
did not find activity type named ...	when adding, etc., an activity type name was specified that does not exist in the model
did not find constraint leaf named ...	when connecting, disconnecting, etc., a constraint name was specified that does not exist

disconnecting constraint failed	tried to disconnect a constraint, but something went wrong and it was not disconnected
inconsistency found in TCN All_Pairs_Shortest_Paths returned false	while propagating the Temporal Constraint Network, an inconsistency was found, meaning that the given set of temporal constraints could not be satisfied
inconsistency found in PCN Propagating:	while propagating the Parameter Constraint Network, an inconsistency was found, meaning that the given set of dependency functions could not be satisfied
incremental list of open constraints is not the same	internal error - send email to aspen@aig and use non-incremental mode (-NI) to bypass
incremental list of open dependencies is not the same	see above
incremental list of violated constraints is not the same	see above
incremental list of violated dependencies is not the same	see above
incremental list of violated timelines is not the same	see above
invalid display type	from interactive mode, the "display" command was given for an unknown object
move called with same time - no effect	no-op
no activity named ...	when moving, deleting, connecting, etc., an activity instance name was specified that does not exist
no activity schema named ...	when adding, etc., an activity type name was specified that does not exist in the model
no constraint named ...	when disconnecting, a constraint name was specified that does not exist
no constraint schema named ...	when connecting, a constraint name was specified that does not exist
no parameter named ...	when changing a parameter value, etc., a parameter name was specified that does not exist
no parameter schema named ...	
no timeline named ...	when displaying, a timeline name was specified that does not exist
no timeline schema named ...	when displaying, a timeline type name was specified that does not exist in the model
removing activity failed ...	tried to remove an activity from the data-base, but something went wrong and it was not removed
temporal network is inconsistent	while propagating the Temporal Constraint Network, an inconsistency was found, meaning that the given set of temporal constraints could not be satisfied
parameter network is inconsistent	while propagating the Parameter Constraint Network, an inconsistency was found, meaning that the given set of dependency functions could not be satisfied

Table 9 Start-up Error Messages

4. Compiling ASPEN

ASPEN is compiled by going to the top-level of the ASPEN directory structure and first sourcing the `setup.csh` file. Then, at the same level, the user types '`make`'. This will create the ASPEN binary executable in the `bin/` directory, and the libraries in the `lib/` directory. In addition, the external function libraries are compiled in the `models/` directories. ASPEN can then be run from this level, or any directory, as long as the environment variables have been set up correctly by the `setup.csh` file. ASPEN can be compiled with several flags:

'`make debug`'	creates a version that can be loaded in the debugger
'`make clean`'	removes all object and executable files, compiles from scratch
'`make purify`'	compiles with purify
'`make fast`'	compiles with optimization and no debug
'`make clean-local`'	in `src/` directory, to save time if you need to clean, but code for the libraries has not changed

Normally, one would compile with '`make fast`', since this will of course create the fastest executable. Because of the extensive optimization used in '`make fast`,' it can take up to three hours to compile depending on the computer. Only the Java GUI and not the X GUI is built by default.

5. Distributing ASPEN

After doing a `'make fast'` on a local machine, if you want to FTP aspen to another machine, you can do `'make dist'` from the top-level aspen directory. This will create the file `'aspen.tar.gz'` which has the following files:

- `setup.csh`
- `bin/aspen`
- `bin/aspenGUI`
- `bin/apfc`
- `bin/ahfc`
- `lib/util-classes.zip`
- `lib/gui-classes.zip`
- `models/test`

This tar file can then be copied to another Sun workstation and decompressed using the following commands:

```
gunzip aspen.tar.gz
tar -xpf aspen.tar
```

Be careful where you gunzip/untar this file after you do `'make dist'`. It creates/overwrites the above set of files from whatever directory you are in when you do the gunzip/untar. The best solution to this problem is to create a new `'aspen'` directory and unpack the files there.

Other distributions available include:

make dist-with-models	Creates a distribution with all available models
make dist-with-headers	Creates a distribution with model header files for compiling
make dist-no-casper	Creates a distribution with everything except CASPER code
make dist-all	Creates a distribution with everything

6. Appendix A – Reserved Words

The following reserved words should not be used as type names in the model file or object names in the initial state file.

Activity	ActivitySchema	add	all
bool	by	capacity	centisecond
change_to	changes_state	char	Conflict
const	constraint	ConstraintLeaf	contained_by
contains	day	decomposition	decomposition_index
default	default_state	delete	dependencies
domain	duration	end	end_time
ending	ends_after	ends_at	ends_before
every	extern	false	float
h	horizon	horizon_duration	horizon_end
horizon_start	hour	infinity	int
Interval	labeled	list	m
mandatory_goal	min_capacity	minute	model
move	must_be	named_reservations	occurs
of	ordered_decomposition	optimize	optional_goal
parameter	permissions	priority	real
repair	RepairState	reservation	reservations
ResolveMethod	resource	s	satisfies
ScheduleDB	second	start	start_time
starting	starts_after	starts_at	starts_before
state_variable	states	string	time
time_scale	timeline_dependencies	times	transition_type
transitions	true	type	usage
use	uses_state	void	w
week	where	with	

7. Appendix B – Related Technical Papers

A. Fukunaga, G. Rabideau, S. Chien, and D. Yan, Feb. 1997, “Toward an Application Framework for Automated Planning and Scheduling,” In *Proceedings of IEEE Aerospace Conference*, 375-386. Washington D.C.: IEEE Computer Society.

S. Chien, D. Decoste, R. Doyle, and P. Stolorz, Apr. 1997. Making an Impact: Artificial Intelligence at the Jet Propulsion Laboratory, *AI Magazine*, 18(1), 103-122.

A. Fukunaga, G. Rabideau, S. Chien, and D. Yan, July 1997, “ASPEN: A Framework for Automated Planning and Scheduling of Spacecraft Control and Operations,” In *Proceedings of the International Symposium on AI, Robotics and Automation in Space (i-SAIRAS)*, Tokyo, Japan.

R. Sherwood, A. Govindjee, D. Yan, G. Rabideau, S. Chien, A. Fukunaga, Oct. 1997. “ASPEN: EO-1 Mission Activity Planning Made Easy,” *In Proceedings of NASA/JPL Workshop on Planning and Scheduling*, Oxnard, CA.

S. Chien, N. Muscettola, K. Rajan, B. Smith, G. Rabideau, Automated Planning and Scheduling for Goal-based Autonomous Spacecraft, *IEEE Intelligent Systems*, September/October 1998, pp. 50-55.

R. Sherwood, A. Govindjee, D. Yan, G. Rabideau, A. Fukunaga, "Using ASPEN to Automate EO-1 Activity Planning," Proceedings of the IEEE Aerospace Conference, 1998, Aspen, CO.

B. Smith, R. Sherwood, A. Govindjee, D. Yan, G. Rabideau, S. Chien, " Representing Spacecraft Mission Planning Knowledge in ASPEN," Proceedings of the AIPS Knowledge Acquisition Workshop (AAAI Technical Report), 1998, Pittsburgh, PA.

Rabideau, G., Chien, S., Estlin, T., Barrett, A., "Working Together: Centralized Command Sequence Generation for Cooperating Rovers," Proceedings of the IEEE Aerospace Conference, Aspen, CO, March 1999.

S. Chien, R. Knight, A. Stechert, R. Sherwood, G. Rabideau, "Integrated Planning and Execution for Autonomous Spacecraft", Proceedings of the IEEE Aerospace Conference, Aspen, CO, March 1999.

Willis, J., Rabideau, G., Wilklow, C., "The Citizen Explorer Scheduling System," Proceedings of the IEEE Aerospace Conference, Aspen, CO, March 1999.

Rabideau, G., Chien, S., Backes, P., Chalfant, G., and Tso, K., "A Step Toward an Autonomous Planetary Rover," Space Technology and Applications International Forum, Albuquerque, NM, 1999.

Sherwood, R., Estlin, T., Chien, S., Rabideau, G., Engelhardt, B., Mishkin, A., Cooper, B., "A Prototype for Ground-based Automated Rover Command Generation," Proceedings of the Second International NASA Workshop on Planning and Scheduling, San Francisco, CA, March 16-18, 2000.

8. Appendix C – Modeling Language Quick Reference

Model Declaration

```
MODEL model_name {
     HORIZON_START = time;
     HORIZON_DURATION = time;
     HORIZON_END = time;
     TIME_SCALE = SECOND|MINUTE|HOUR|DAY|WEEK;
     TIME_ZONE = LOCAL|UTC;
     TIME_FORMAT = SLASH|TEE|CALENDAR;
};
```

Activity Declaration

```
ACTIVITY act_name {
     INT|BOOL|REAL|STRING param_name = value(s);
     DURATION = value(s);
     START_TIME = param_name | value(s);
     END_TIME = param_name | value(s);
     PRIORITY = value(s);
     PERMISSIONS = value(s);
     DEPENDENCIES = depend_list;
     TIMELINE_DEPENDENCIES = depend_list;
     RESERVATIONS = reservation_list;
     NAMED_RESERVATIONS = named_reservation_list;
     DECOMPOSITIONS = decomposition_list;
     CONSTRAINTS = constraint_list;
};

values = (value, value, …) | [value, value]
value = int | bool | float | string
depend_list = depend, depend, …
depend = param->param | param<-param | param<->param | param<-f(param)
activity_list = activity, activity, …
activity = act_name WITH (depend_list)
reservation_list = reservation, reservation, …
reservation = timeline_name USE|MUST_BE|CHANGE_TO param|value(s)
named_reservation_list = named_reservation, named_reservation, …
named_reservation = reservation_name WITH (depend_list)
decomposition_list = decomposition OR decomposition OR …
decomposition = (activity_list WHERE constraint_list)
constraint_list = (constraint AND|OR constraint …);
constraint =
 STARTS_BEFORE|STARTS_AFTER|ENDS_BEFORE|ENDS_AFTER|CONTAINS|CONTAINED_BY
 START|END|ALL OF act_name WITH (depend_list) BY range(s);
```

Resource Declaration

```
RESOURCE resource_name {
     TYPE = ATOMIC|DEPLETABLE|NONDEPLETABLE;
     CAPACITY = value;
     MIN_VALUE = value;
};
```

State Declaration

```
STATE_VARIABLE sv_name {
     STATES = values;
     TRANSITIONS = value_pairs;
     DEFAULT_STATE = value;
};

value_pairs = (value->value, value->value, …)
```

Reservation Declaration

(reservation definitions are no longer necessary for most situations)

```
RESERVATION reservation_name {
     INT|BOOL|REAL|STRING param_name = value(s);
     RESOURCE = resource_name | STATE_VARIABLE = sv_name;
     USAGE|USES_STATE|CHANGES_STATE = value(s);
};
```

Parameter Declaration

```
PARAMETER type param_name {
     DOMAIN = value(s);
     DEFAULT = value;
};
```

9. Appendix D – Score Functions

Total score for *n* preferences:

$$score = \frac{\sum_{i=1}^{n} score(p_i) * weight(p_i)}{\sum_{i=1}^{n} weight(p_i)}$$

Score functions for individual preferences:

v	= value of the parameter
lb	= lower bound of preference range
ub	= upper bound of preference range
cv	= preferred value for centered preferences
MAX	= maximum possible score (constant)

Linearly more:

$$score(v, lb, ub) = \begin{cases} 0.0 & when(v < lb) \\ MAX & when(v > ub) \\ MAX * \left(\frac{v - lb}{ub - lb}\right) & otherwise \end{cases}$$

Linearly less:

$$score(v, lb, ub) = \begin{cases} MAX & when(v < lb) \\ 0.0 & when(v > ub) \\ MAX - MAX * \left(\frac{v - lb}{ub - lb}\right) & otherwise \end{cases}$$

Linearly centered:

$$score(v, lb, ub, cv) = \begin{cases} 0.0 & when(v < lb \parallel v > ub) \\ MAX * \left(\frac{v - lb}{cv - lb}\right) & when(v < cv) \\ MAX - MAX * \left(\frac{v - cv}{cv - lb}\right) & when(v \geq cv) \end{cases}$$

Exponentially more:

$$score(v, lb, ub) = \begin{cases} 0.0 & when(v < lb) \\ MAX - \left(MAX * e^{(lb - v)} \right) & otherwise \end{cases}$$

Exponentially less:

$$score(v, lb, ub) = \begin{cases} MAX & when(v < lb) \\ MAX * e^{(lb - v)} & otherwise \end{cases}$$

Exponentially centered:

$$score(v, lb, ub, cv) = \begin{cases} 0.0 & when(v < lb \parallel v > ub) \\ MAX * e^{(v - cv)} & when(v < cv) \\ MAX * e^{(cv - v)} & when(v \geq cv) \end{cases}$$

Stepwise more:

$$score(v, lb, ub) = \begin{cases} 0.0 & when(lb = ub \neq v) \\ MAX & when(lb = ub = v) \\ MAX * \left(\dfrac{indexof(v) - indexof(lb)}{indexof(ub) - indexof(lb)} \right) & otherwise \end{cases}$$

Stepwise less:

$$score(v, lb, ub) = \begin{cases} 0.0 & when(lb = ub \neq v) \\ MAX & when(lb = ub = v) \\ MAX - MAX * \left(\dfrac{indexof(v) - indexof(lb)}{indexof(ub) - indexof(lb)} \right) & otherwise \end{cases}$$

Stepwise centered:

$$score(v, lb, ub, cv) = \begin{cases} 0.0 \} when(lb = ub = cv \neq v) \\ MAX \} when(lb = ub = cv = v) \\ MAX * \left(\dfrac{indexof(v) - indexof(lb)}{indexof(cv) - indexof(lb)} \right) \} when(indexof(v) < indexof(cv)) \\ MAX - MAX * \left(\dfrac{indexof(v) - indexof(cv)}{indexof(ub) - indexof(cv)} \right) \} when(indexof(v) \geq indexof(cv)) \end{cases}$$

10. Index